ESD-TR-76-288, Vol. I

MTR-3178, Vol. I

COMPUTER PROGRAM SPECIFICATION FOR SECURITY KERNEL FOR PDP-11/45

APRIL 1978

Prepared for

DEPUTY FOR TECHNICAL OPERATIONS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Bedford, Massachusetts

Project No. 7070

Prepared by

THE MITRE CORPORATION
Bedford, Massachusetts
Contract No. F19628-75-C-0001

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

BRIAN W. WOODRUFF, Captain, USAF
Project Engineer

DONALD P. ERIKSEN
Project Engineer

JOHN T. HOLLAND, Lt Col, USAF
Chief, Techniques Engineering Div

FOR THE COMMANDER

STANLEY P. DERESKA, Colonel, USAF
Director, Computer Systems Engineering
Deputy for Technical Operations

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE — READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER ESD-TR-76-288, Vol. I	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) COMPUTER PROGRAM SPECIFICATION FOR SECURITY KERNEL FOR PDP-11/45		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER MTR-3178, Vol. I
7. AUTHOR(s) S.R. Harper		8. CONTRACT OR GRANT NUMBER(s) F19628-75-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS The MITRE Corporation Box 208 Bedford, MA 01730		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Project No. 7070
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy for Technical Operations Electronic Systems Division, AFSC Hanscom Air Force Base, MA 01731		12. REPORT DATE APRIL 1978 13. NUMBER OF PAGES 336
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

COMPUTER SECURITY TECHNOLOGY
PDP-11/45
SECURITY KERNELS

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report presents the Type C5, Computer Program Product Specification for the Security Kernel for the PDP-11/45. It specifies the configuration information that fully describes the Security Kernel as an established program product. A detailed description of each individual function of the program is given. Also included are the requirements which provide the basis for development of verification procedures

(over)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract (continued)

and a section of informal notes for the potential user. Flow charts are included in an addenda to this report for the convenience of the reader. A complete listing of the Security Kernel program is presented in Section 10, Volume II.

ACKNOWLEDGMENT

This report has been prepared by The MITRE Corporation under Project No. 7070. The contract is sponsored by the Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Massachusetts.

This report is a compilation of contributions from the Project 7070 Staff Members, notably: Grace H. Nibaldi, John A. Larkins, Daniel F. Stock, John C. C. White and Alan R. Chen. Special acknowledgements are extended to John L. Mack for his preparation of Sections IV and V and to W. Lee Schiller for his "Design and Specification of a Security Kernel for the PDP-11/45" on which much of the material in this report is based.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. SCOPE

This specification establishes the requirements for complete identification of the ESD/MITRE Security Kernel Computer Program Product (SKCPP) for the PDP-11/45, henceforth to be referred to as the Security Kernel. The purpose of the specification is to present the information necessary to understand the Security Kernel for use, either for further investigation, or for implementation in a similar hardware environment.

The Security Kernel is the software, when installed and running properly on the PDP-11/45 hardware, that provides the facilities to control access of active system elements (subjects) to units of information (objects) within the computer system, thus providing the basis for secure computer systems on the PDP-11/45.

2. REFERENCED DOCUMENTS

The following documents, of exact issue shown, form a part of this specification to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this specification, the contents of this specification shall be considered a superseding requirement.

2.1 Government Documents

a. Bell, D. E., L. J. LaPadula, "Secure Computer Systems", ESD-TR-73-278, Volumes I, II, III, November 1973 - April 1974 (AD770768, AD771543, AD780528).

b. DoD 5200.1, DoD Information Security Program, June 1972.

c. DoD 5200.1-R, Regulations Governing the Classification, Downgrading, Declassification and Safeguarding of Classified Information, November 1973.

d. Schiller, W. L., "Design and Specification of a Secure Kernel for the PDP-11/45", ESD-TR-75-69, May 1975 (ADA011712).

e. Stork, D. F., "Downgrading in a Secure Multilevel Computer System: The Formulary Concept", ESD-TR-75-62, May 1975 (ADA011696).

2.2 Non-Government Documents

a. Clark, B. L., F. J. B. Ham, "The Project SUE System Language Reference Manual", Computer Systems Research Group, University of Toronto, September 1974.

b. KT11-C Memory Management Unit Maintenance Manual, Digital Equipment Corporation, Maynard, MA., 1973.

c. PAL-11R Assembler, Digital Equipment Corporation, Maynard, MA., 1973.

d. Parnas, D. L., "A Technique for Software Module Specification with Examples", Communications of the ACM, Volume 15, No. 5, 330-336, May 1972.

e. PDP-11 Peripherals Handbook, Digital Equipment Corporation, Maynard, MA., 1973.

f. PDP-11/45 Processor Handbook, Digital Equipment Corporation, Maynard, MA., 1973.

g. User's Manual for Delta 5000 Family of Video Display Terminals, Delta Data Systems, Cornwells Heights, PA., August 1974.

3. REQUIREMENTS

The Security Kernel as described in this specification operates on a Digital Equipment Corporation PDP-11/45 with memory management unit with 128K bytes of core (reference 2.2b, e, and f) and a mass storage device in the form of an RF11 disk with 512K bytes available. The other peripherals associated with this PDP-11/45 are two Delta Data Systems Delta 5000 scopes (reference 2.2g), an ASR-33 teletype and a Digital Equipment Corporation DECwriter (reference 2.2e). Only minor modifications to the Security Kernel's code would be required to add, delete, or change the peripherals. The aforementioned storage, however, is considered to be the minimal requirement for effective execution.

The output medium used for transfer of the load module from the IBM 360 environment, where the Project SUE System and the PDP-11 Cross Assembler both execute, to the PDP-11/45 is a 9 track magnetic tape. The Digital Equipment Corporation TM11 magnetic tape system (reference 2.2e) is used to load the 9 track magnetic tape into the PDP-11/45.

A reference monitor is fundamental to a secure computer system. The reference monitor is that portion of a computer's hardware and software which enforces the authorized access relationships between subjects and objects. Subjects are entities such as a user or a process that seek to gain access to objects, while objects are entities such as data, programs, and peripheral devices to which access must be gained in the course of the system's use. The RF11 disk is the only DMA (direct memory access) device the Security Kernel handles. It has complete control over what is written and what is read since it has sole write access to the disk status register, which contains disk address, memory address, mode, and word count.

The memory management unit (MMU) option of the PDP-11/45 provides the hardware facilities that make it a suitable base for a secure system. The MMU checks all references to memory and supports enforcement of three access modes -- read/write, read, and no access protection. It also provides a hierarchy of three domains (or modes) of execution -- kernel, supervisor, and user. The hardware affects the hierarchical ordering of domains by permitting the execution of certain machine instructions in the kernel domain only, and restricting the manner in which the instructions which pass control from domain to domain operate. The MMU performs dynamic address translation; each time an effective address is generated during instruction execution, it is treated as a 16-bit virtual address and translated into an 18-bit physical address before

reference to main memory is made. Figure 1 portrays the dynamic address translation performed by the MMU. The translation is controlled by the contents of a set of eight segmentation registers, one set for each domain of execution.

The Security Kernel is the software portion of the reference monitor, the control and certification of which is essential to achieve a basic security module. It protects itself by insuring that its own procedures are the only ones that execute in kernel mode, and by executing interpretively all attempts to access the Security Kernel data base that originate with processes executing outside the kernel.

Interpretive execution, within the kernel, of access attempts permits objects accessed in the kernel domain to be portions of segments, whereas a directly accessed object outside the kernel must be coextensive with a single segment (reference 2.1d); a segment is described as a collection of from 64 to 8194 bytes of contiguous virtual memory, which is uniquely identified, and which may be inserted into an appropriately sized and protected region of main memory.

The identification of a subject recognized by the Security Kernel is an ordered pair, whose components are a process and a domain. A process, in turn, is identified by the kernel as operating on behalf of a user/project pair. Information that describes processes is contained in data structures that are accessed in the kernel domain.

The Security Kernel software, operating in the PDP-11/45 with MMU, provides a non-random access, segmented virtual memory. The segmented virtual memory is organized into a tree-like directory hierarchy. The set of all segments in the hierarchy is the system space (SS). By virtue of security attributes and a security policy, most users of the system will not be able to access all of the segments in SS. The subset that a particular user may access is called a virtual space (VS). When a user process starts execution in a system that incorporates the Security Kernel, it has a virtual machine executing on its behalf. This process will have an address space of segments constrained in size by certain design and implementation parameters. The process's address space is called the working space (WS), and is always a subset of the user's VS. Finally, to permit a process direct access to all segments in its WS would severely constrain the size of the WS (to the number of descriptors available). For that reason another space, access space (AS), is added. AS, a subset of WS, represents the segments that a process can directly address because it has descriptors

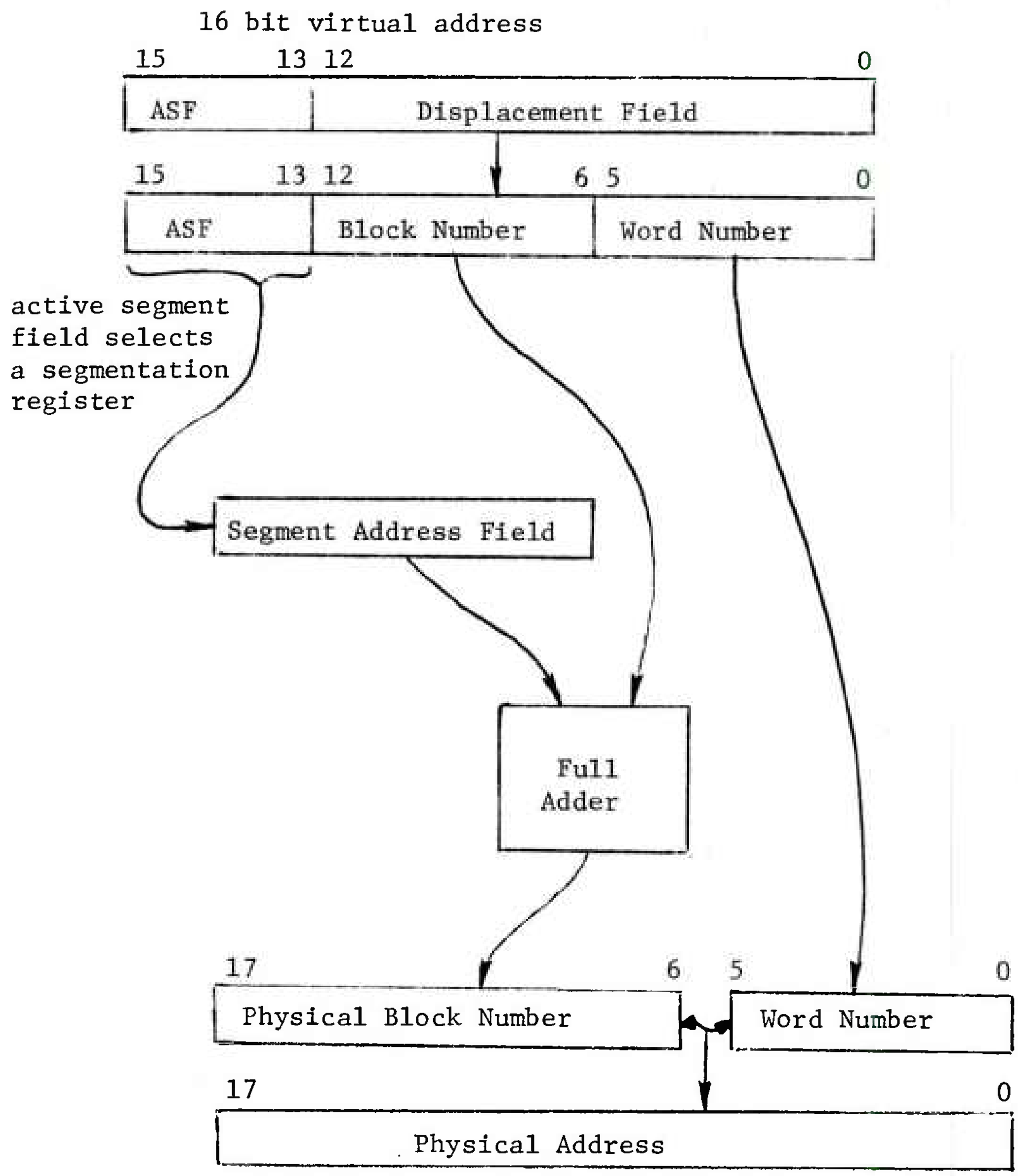


Figure 1. Dynamic Address Translation

available. A process, therefore, may frequently remove a segment from AS to make room for another, without necessarily removing the segment from WS. Figure 2 shows these space relationships.

3.1 Functional Allocation Description

a. Control Entry and Exit functions accept and check parameters furnished by the user.

b. Directory Manipulation functions change the security state of the system by creating or deleting segments, and by adding or deleting elements in a segment's control list.

c. Segment Accessing functions move segments into and out of a process's working space.

d. Process Cooperation functions allow the sequential processes that coexist in the physical computer system to cooperate in performing computations.

e. Process Control functions select a particular process to which the CPU will be allocated, while saving information pertaining to other processes.

f. Privileged functions establish certain system conditions, and may only be invoked by trusted subjects.

Of the forty-two CPC's, eighteen are externally callable, that is, they are called by a process operating outside the perimeter of the Security Kernel (external to the kernel domain). These components are accessed by way of kernel commands, which include the symbolic name of the function along with the appropriate arguments.

The externally callable functions are listed below according to their functional uses.

Directory Manipulation

- create
- delete
- give
- rescind
- read directory

Segment Accessing

- get write
- get read

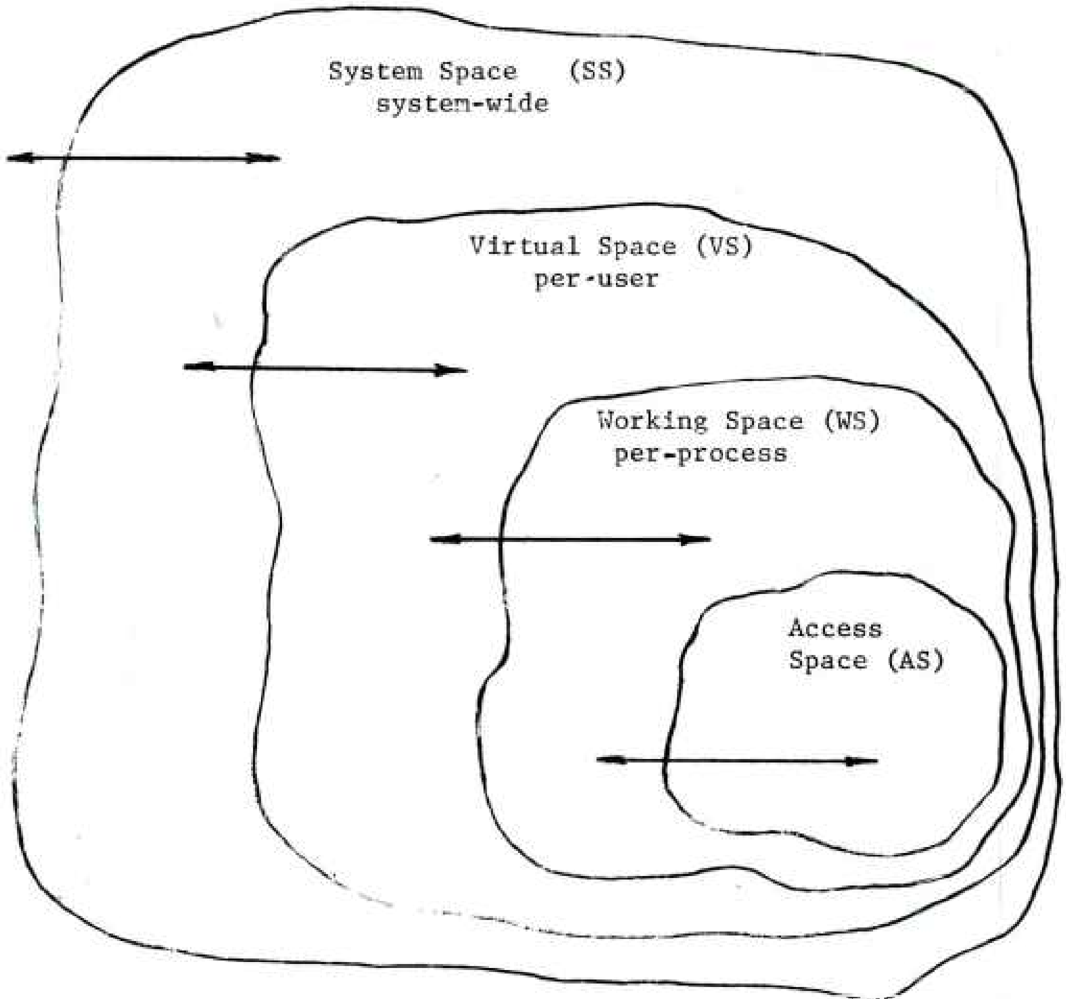


Figure 2. Spaces

- release
- enable
- disable
- outer P
- outer V
- send interprocess communication
- receive interprocess communication

Process Control

- stop process

Privileged

- start process
- change object
- initialize hierarchy

The remaining twenty-four CPC's are called once the kernel domain has been entered (internal to the kernel domain). They are called either directly or indirectly by the externally callable components and are invisible outside the Security Kernel.

The internally callable functions are listed below according to their functional uses.

Control Entry and Exit

- gate
- parameter checker

Directory Manipulation

- delete segment
- disk allocation
- free disk
- search out and destroy descriptors
- get directory
- write directory

Segments Accessing

- directory search
- connect
- activate
- deactivate
- prehash
- hash
- load segment descriptors
- swapin
- swapout

. initialize segment
. disk input/output

Process Cooperation

. P
. V

Process Control

. sleep
. run
. swap

The kernel function gate (GATE) is the sole entry and exit point into and out of the kernel domain. When a user process requests an externally callable user level function, a trap is generated (reference 2.2f) and as a result of this trap GATE is entered. Upon entry, the supervisor's stack is accessed for parameter passing. The parameter checker (PCHECK) is then invoked by GATE and the parameters associated with the user requested function are checked to assure that they are within the acceptable ranges. If all parameters are within bounds PCHECK invokes the user requested function. Once the user requested function has performed its task, the return code (RC) is passed through PCHECK to GATE. GATE accesses the supervisor's stack, places the RC on the supervisor's stack and the kernel domain is exited. The only access route through the security perimeter into kernel domain is through GATE, which is also the only exit route.

The Security Kernel supports five processes which operate on behalf of the user (see Section 3.). The interrupt handlers for these I/O devices are handled within GATE. It saves the contents of the current process's registers; the PC (program counter) and PSW (process status word) (reference 2.2f) of the interrupt vector become the new process PC and PSW. The new current process is made available for servicing by a call to V which increments the semaphore on this process's I/O segment. The general purpose registers, the PC, and the PSW are restored to what they contained before the interrupt. For a more detailed description of GATE refer to paragraph 3.2.1.

The following subparagraphs provide more detail on the Security Kernel computer program components, the functional areas in which they fall, their symbolic code, and the major function they perform.

To avoid confusion an explanation of the release function is in order here. The symbolic code of the release function changes once the kernel domain has been entered. Externally (outside kernel domain) the function is known as RELEASE to match the mathematical model (reference 2.1a). Internally (inside kernel domain) the function is known as DCONNECT, as it is the inverse of the function CONNECT.

3.1.1 Entering and Exiting Kernel Domain

Two functions are provided to accept and check parameters furnished by the user. These functions are always invoked when an external call is made to the Security Kernel or upon exiting from kernel domain.

3.1.1.1 Accepting and Checking Parameters

3.1.1.1.1 Internal Functions

Gate	GATE	the entry and exit point into and out of the kernel.
Parameter Checker	PCHECK	assures that all user input parameters are within bounds.

3.1.2 Directory Manipulation Functions

A set of functions is provided for manipulating the attributes of segments. These functions change the security state of the system by creating and deleting segments, and by adding and deleting elements to/from a segment's access control list (ACL). The common security requirement for all functions that modify segment attributes is that the modifying process currently have write access to the segment's parent directory.

3.1.2.1 Creating and Deleting Segments

3.1.2.1.1 External Functions

Create	CREATE	creates a segment inferior to a specified directory segment.
Delete	DELETE	deletes an existing segment and any segments subordinate to it.

3.1.2.1.2 Internal Functions

Delete Segment	DELETSEG	deletes an empty directory or a data segment.
Disk Allocation	DALLOC	allocates space on the disk as segments are created.
Free Disk	DFREE	frees space on the disk as segments are deleted.

3.1.2.2 Giving and Rescinding Access

3.1.2.2.1 External Functions

Give	GIVE	adds an ACL element to a segment's ACL chain.
Rescind	RESCIND	removes an ACL element from a segment's ACL chain.

3.1.2.3 Directory Support Functions

3.1.2.3.1 Internal Functions

Search Out and Destroy Descriptors	SOADD	removes a segment from the address space of all processes that currently have access to that segment.

3.1.2.4 Reading Directories

3.1.2.4.1 External Functions

Read Directory	READIR	provides interpretive read access to an object that is in a process' address space.

3.1.2.5 Directory Checking

3.1.2.5.1 Internal Functions

Get Directory	GETDIR	assures that a directory segment to be written is in main memory and accessible.

Write Directory	WRITEDIR	checks that a segment is a directory and if so, that the current process has write access to it.

3.1.3 Segment Accessing

There are functions provided for moving segments into and out of a process's working space (WS) and functions that support the implementation of access space (AS). The function that changes a process's WS changes the state of the system with respect to security, whereas the functions that change AS are only changing the representation of the current security state. A process can directly address only those segments in its WS that are also in its AS because of hardware segmentation register constraints. WS is defined, for security purposes, to be the address space of a process.

3.1.3.1 Getting and Releasing Access

3.1.3.1.1 External Functions

Get Write	GETW	moves a segment into a process's WS in write mode if all requirements are satisfied.
Get Read	GETR	moves a segment into a process's WS in read mode if all requirements are satisfied.
Release	DCONNECT	removes a segment from a process's WS (known externally as RELEASE, internally as DCONNECT).

3.1.3.1.2 Internal Functions

Directory Search	DSEARCH	searches an ACL chain looking for an ACL element that applies to the invoking process.

3.1.3.2 WS Support Functions

3.1.3.2.1 Internal Functions

Connect	CONNECT	connects a process to a segment in its WS.

Activate	ACT	copies the directory entry into an ASTE (active segment table entry) and initializes other fields in the ASTE.
Deactivate	DEACT	removes a segment from the AST (Active Segment Table).
Prehash	PREHASH	computes an index into the hash table.
Hash	HASH	converts a disk address (which uniquely identifies a segment) to an ASTE#, using a hash table.

3.1.3.3 Enabling and Disabling Access

3.1.3.3.1 External Functions

Enable	ENABLE	moves a segment in a process's WS into its AS.
Disable	DISABLE	removes a segment from a process's AS and frees a segmentation register.

3.1.3.3.2 Internal Functions

Load Segment Descriptor	LSD	constructs segment descriptors and inserts them into descriptor registers.

3.1.3.4 AS Support Functions

3.1.3.4.1 Internal Functions

Swap In	SWAPIN	swaps a segment into main memory.
Swap Out	SWAPOUT	removes a segment from main memory.
Initialize Segment	INITSEG	initializes a segment in memory.
Disk Input/Output	DISKIO	performs a disk I/O operation.

3.1.4 Process Cooperation

Mechanisms are provided to allow the sequential processes that coexist in the physical computer system to cooperate in performing computations. These mechanisms are used within the Security Kernel to insure its correct operation. The Security Kernel provides functions that allow these mechanisms to be used in an arbitrary manner subject only to security constraints. These functions do not change the security states of the system. They provide interpretive access to objects as permitted by the current state, and since they can cause the execution state of a process to change, they modify the representation of the current state.

3.1.4.1 Synchronization Primitives

3.1.4.1.1 External Functions

Outer P Outer V	OUTERP OUTERV	when a P or V is performed externally,OUTERP and OUTERV perform implementation and security checks.

3.1.4.1.2 Internal Functions

P and V	P and V	allows users to coordinate the modification of shared segments and to synchronize with their terminal I/O. P and V are the standard synchronization primitives.

3.1.4.2 Interprocess Communication

3.1.4.2.1 External Functions

Send Interprocess Communication	IPCSEND	allocates an IPC element, fills it in and adds it to the queue of elements waiting to be received.
Receive Interprocess Communication	IPCRCV	removes the data from an IPC element and puts the element on the free chain.

3.1.5 Process Control

A set of functions are provided for selecting a particular process to run, and for preparing and saving information pertaining to the process that currently has the CPU allocated to it and the next ready process.

3.1.5.1 Creating and Deleting Processes

3.1.5.1.1 External Functions

Stop Process	STOPP	relinquishes a user's ownership of a process.

3.1.5.1.2 Internal Functions

Sleep	SLEEP	schedules another process, in a round robin fashion.
Run	RUN	saves information on the current process and prepares to run the next process.
Swap	SWAP	establishes the next process's address space.

3.1.6 Privileged Functions

Three kernel functions are invoked by the executive process only.

3.1.6.1 Trustworthy Process Functions

3.1.6.1.1 External Functions

Change Object	CHANGEO	performs a change in classification of a non-directory segment.
Initialize Hierarchy	INITH	sets up the initial directory structure at initialization time.

Start Process STARTP initializes a new process.

3.2 Functional Description

This paragraph contains the detailed technical descriptions of the computer program components identified in paragraph 3.1 of this specification. The instruction listing contained in Section 10 specifies the exact configuration of the Security Kernel.

Most of the computer program components of the Security Kernel are written in Project SUE System Language (reference 2.2a). DALLOC, DFREE, DISKIO, LSD, and SWAP, which deal exclusively with disk space and hardware registers, are written in PAL-11, the PDP-11/45 assembler language (reference 2.2c).

The SUE language facilitates highly readable structured programs and data. One of its features is a macro facility which permits text substitution with parameters, but no compile-time computation. That is, the macro name is textually replaced by its macro body at compile time, thus eliminating repetitive coding and branching. One of the ways the Security Kernel makes use of this feature is in producing a more efficient and more exact calling sequence.

Each externally callable function has associated with it a macro whose name begins with 'K'; e.g., KCREATE is the macro associated with the CREATE function. These macros are located in the Context Block (reference 2.2a) NOFORN (Section 10, pages 2 through 10) which contains definitions relevant to all kernel users but is external to the security perimeter of the Security Kernel. The effect of these macros is to generate code which will place the parameters of the requested function on the supervisor's stack and force an interrupt. As a result of the interrupt a branch into the kernel domain is effected. An example of the calling sequence is as follows.

A user process requests the externally callable function CREATE by using the Kernel Command CREATE along with its appropriate parameters. The macro KCREATE generates code which places the input parameters on the supervisor's segmentation register Ø stack and forces an interrupt. As a result of this interrupt a branch into GATE is effected. GATE accesses the supervisor's stack through kernel segmentation register 3 (KSR3). The parameters, which are

now residing in KSR3, are passed to PCHECK for checking. If at this point the parameters are within the acceptable ranges, PCHECK invokes the CREATE function.

Many of the kernel functions set the value of a per process return code (RC). The security attributes of the RC object are equal to those of the process. In general, Security Kernel functions set RC to indicate whether or not they were called correctly. A few functions use RC to return additional information to the user. Each process can always observe its own and only its own RC object. Once the called function has performed its task, it passes the RC through PCHECK to GATE. GATE then performs the inverse of its entry sequence, that is, it gains write access to the supervisor's stack through KSR3 and places the RC on it.

Another feature of the Project SUE System Language (reference 2.2a) is the use of Inline. Inline inserts arbitrary machine code inline at compile time. The parameters are quantities that cause code to be placed in the machine instruction format.

Figure 3 shows the intended interpretation of the external kernel function parameters and internal kernel variables. TCP (the current process) is an internal Security Kernel variable that indicates which process is currently bound to the CPU. It is part of the mechanism for implementing a distributed kernel and prevents users of the Security Kernel from forging their identity.

The individual CPC's are described in the following 3.2 subparagraphs. Figure 3a identifies the form in which each CPC is presented.

External Security Kernel function parameters

seg#	segment number of a segment in a process's address space (WS)
offset	identification of an entry within a directory
class	a classification
cat	a category set
type	DATA or DIRECTORY
size	size of a segment in blocks
mode	WRITE, READ, or NO
user_id	user identification
project_id	project identification
reg#	identification of a segmentation register
process#	identification of a process
block#	main memory address of a segment

Internal Security Kernel "variables"

TCP	the current process
aste#	pointer to an AST entry
daste#	aste# for a segment known to be a directory
acle#	pointer to an ACL element
smfr#	pointer to a semaphore
ipce#	pointer to an IPC element

Figure 3. Intended Interpretations

3.2.n CPC Title (symbolic code)

Identification of the level of the SKCPP function, the level of functions that call it, the level of functions that it calls and the language in which it is written.

3.2.n.1 Description

A description of the internal requirements of the CPC. A formal specification is also presented.

3.2.n.2 Flow Charts

Flow charts are presented for the PAL-11 CPCs exclusively as it is felt that the highly readable structure of the SUE language makes flow charting of those CPCs redundant.

3.2.n.3 Interfaces

The relationships of the CPCs to each other are presented here by listing those CPCs that call and those CPCs called by this specific CPC.

3.2.n.4 Data Organization

The relationships of the CPC to the data base structures are presented here. Global references, function parameters local references, and constants are listed.

3.2.n.5 Limitations

Any known limitations of the CPC are presented here.

3.2.n.6 Listing

A complete listing of the Security Kernel is provided in Section 10; however, this subparagraph includes the DATA BLOCK and PROGRAM BLOCK of the specific CPC.

Figure 3a. CPC Write-Up Form

3.2.1 Gate (GATE)

The Gate CPC, GATE, is a kernel level internal SKCPP function that is invoked as the result of a user level program requesting an externally callable Security Kernel function. This CPC is unique as it is the sole entry and exit point into and out of the domain of the Security Kernel, that is, an externally callable function can only be reached through GATE and all return codes (if applicable) are passed to the user through GATE. GATE directly calls the kernel level internal function PCHECK, which in turn, directly calls the user requested function. The other function that GATE calls is the kernel level internal function V. It is written in Project SUE Language.

3.2.1.1 Description

GATE provides the only entry and exit point to and from the Security Kernel. It performs the KERNEL_ENTRY macro which pushes the supervisor's stack and registers onto the kernel's stack. If the logical and of PSW and PREV_MODE_MASK does not equal PREV_MODE_SUPERV, the call was not made from supervisor mode and GATE ignores it. Otherwise, the kernel must access the supervisor's stack through KSR3 to get its parameters, so it assigns to KSDR3 and KSAR3 the values of SDRØ and SARØ. GATE then calls PCHECK to check the parameters and to call the function requested, and sets RC to the value PCHECK returns. It then assigns to KSR3 and KRC the values of SRØ and RC, regaining access to the supervisor's stack and inserting the return code. GATE then invokes the KERNEL_EXIT macro to restore supervisor registers RØ to R6 from its stack and return. GATE handles interrupts by invoking KERNEL ENTRY to save the contents of the current process's registers; the PS and PSW of the interrupt vector become the new process's PC and PSW. GATE calls V to increment the semaphore on the new current process, so it can be serviced. KERNEL_EXIT is then invoked, restoring the general purpose registers, the PC, and the PSW with what they contained before the interrupt.

```
Function:  GATE
Parameters:  GATE(function_code,seg#,offset,class,cat,type,
             size,mode,user_id,project_id,reg#,process#,
             block#)
Effect:
IF PSW & PREV_MODE_MASK = PREV_MODE_SUPERV;
THEN:  PCHECK(function_code,seg#,offset,class,cat,type,size,
       mode,user_id,project_id,reg#,process#,block#);
END;
```

3.2.1.2 N/A

3.2.1.3 Interfaces

As a direct result of a user level external program requesting an externally callable Security Kernel function, the macro associated with the requested function is invoked. The macro generates code which places the user entered parameters on the supervisor's stack and causes an interrupt. As a direct result of this interrupt, entry into the Security Kernel domain is reached.

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
see above paragraph	PCHECK
	V

3.2.1.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function GATE. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Functional Parameters	Local References
SDRØ	FUNCTION_CODE	KSDR3
SARØ	SEG#	KSAR3
PSW	OFFSET	RC
	CLASS	
	CAT	
	TYPE	
	SIZE	
	MODE	
	USER_ID	
	PROJECT_ID	
	REG#	
	PROCESS#	
	BLOCK#	

Constants

DECW_PROCESS#
DISK_SMFR
PREV_MODE_MASK

SCOPE1_PROCESS#
SCOPE2_PROCESS#

A short discussion of the two Context Blocks CONTEXT NOFORN and CONTEXT KERNEL seems appropriate at this point. The Context Block NOFORN contains definitions relevant to all kernel users and the Security Kernel supportive software. The Context Block KERNEL contains definitions of the Security Kernel's virtual address space. These, of course, reside outside the security perimeter of the Security Kernel. Refer to Section 10 for listing of both Context Blocks.

DATA GATE contains the procedure declarations for all externally callable functions and all but six of the internally callable functions. Those six, DELETSEG, DALLOC, DFREE, GETPIR, INITSEG, and SWAP, are called by only one function and are declared in the Data Block of that function.

3.2.1.5 Limitations

Entry into kernel domain is only affected if the call is made from supervisor domain. The return code (RC) is that which the requested function has passed through PCHECK.

3.2.1.6 Listing

```
DATA GATE;

     /*    MUST CHECK INITIALIZATION OF INTERRUPT VECTORS IN STARTUP EVERY TIME A        *
      * CHANGE IS MADE TO GATE (DATA OR PROGRAM)!                                        */

     MACRO KERNEL_ENTRY;
          INLINE(MFPIR6);
          INLINE(MOV, 0, 5, 4, 6);
          INLINE(MOV, 0, 4, 4, 6);
          INLINE(MOV, 0, 3, 4, 6);
          INLINE(MOV, 0, 2, 4, 6);
          INLINE(MOV, 0, 1, 4, 6);
          INLINE(MOV, 0, 0, 4, 6);
          INLINE(MOV, 0, 6, 0, 4);
          INLINE(MOV, 0, 6, 0, 5);
          INLINE(MOV, 2, 7, 4, 6, "800A");
          INLINE(SUB, 2, 7, 0, 6, 4)
     END MACRO;

     MACRO KERNEL_EXIT;
          INLINE(ADD, 2, 7, 0, 6, 6);
          INLINE(MOV, 2, 6, 0, 0);
          INLINE(MOV, 2, 6, 0, 1);
          INLINE(MOV, 2, 6, 0, 2);
          INLINE(MOV, 2, 6, 0, 3);
          INLINE(MOV, 2, 6, 0, 4);
          INLINE(MOV, 2, 6, 0, 5);
          INLINE(MTPIR6);
          INLINE(RTI)
     END MACRO;
```

```
DECLARE
     WORD (RC);

/*    EXTERNEL KERNEL FUNCTIONS                                              */

DECLARE
     PROCEDURE (STOPP),
     PROCEDURE ACCEPTS (WORD) (DISABLE),
     PROCEDURE ACCEPTS (WORD, WORD) (DCONNECT),
     PROCEDURE ACCEPTS (WORD, WORD, WORD) (IPCSEND),
     PROCEDURE RETURNS (WORD) (IPCRCV),
     PROCEDURE ACCEPTS (WORD) RETURNS (WORD) (OUTERP, OUTERV),
     PROCEDURE ACCEPTS (WORD, WORD) RETURNS (WORD) (DELETE, GETW, GETR, ENABLE,
          READIR),
     PROCEDURE ACCEPTS (WORD, WORD, WORD) RETURNS (WORD) (INITH),
     PROCEDURE ACCEPTS (WORD, WORD, WORD, WORD) RETURNS (WORD) (RESCIND, CHANGEO),
     PROCEDURE ACCEPTS (WORD, WORD, WORD, WORD, WORD) RETURNS (WORD) (GIVE),
     PROCEDURE ACCEPTS (WORD, WORD, WORD, WORD, WORD, WORD) RETURNS (WORD) (CREATE,
          STARTP);

/*    INTERNAL KERNEL FUNCTIONS                                              */

DECLARE
     PROCEDURE (SLEEP),
     PROCEDURE ACCEPTS (WORD) (DEACT, P, V, SWAPIN, SWAPOUT, RUN),
     PROCEDURE ACCEPTS (WORD, WORD) (SOADD),
     PROCEDURE ACCEPTS (WORD, WORD, WORD) (LSD),
     PROCEDURE ACCEPTS (WORD, WORD, WORD, WORD) (DISKIO),
     PROCEDURE RETURNS (WORD) (PCHECK),
     PROCEDURE ACCEPTS (WORD) RETURNS (WORD) (WRITEDIR, HASH, PREHASH),
     PROCEDURE ACCEPTS (WORD, WORD, WORD) RETURNS (WORD) (DSEARCH, CONNECT),
     PROCEDURE ACCEPTS (WORD, WORD) RETURNS (WORD) (ACT);


PROGRAM GATE;

     /*    PUSH SUPERVISOR R6 AND R0-R5 ONTO KERNEL'S STACK                  */

     KERNEL_ENTRY;

     /*    IGNORE CALL IF NOT MADE FROM SUPERVISOR MODE                      */

     IF (PSW & PREV_MODE_MASK) = PREV_MODE_SUPERV;
       THEN:

          /*    ACCESS SUPERVISOR'S STACK THROUGH KSR3                       */

          KSDR3 := SDR0;
          KSAR3 := SAR0;

          /*    CALL PARAMETER CHECKER - WHO IN TURN CALLS THE REQUESTED FUNCTION */

          RC := PCHECK;

          /*    REGAIN ACCESS TO SUPERVISOR'S SR0 STACK AND INSERT RETURN CODE */

          KSDR3 := SDR0;
          KSAR3 := SAR0;
          KRC := RC;
     END;

     /*    RESTORE SUPERVISOR'S REGISTERS AND RETURN                         */

     KERNEL_EXIT;

     /*    INTERRUPT ENTRY POINTS                                            */

     /*    DISK                                                              */
```

```
KERNEL_ENTRY;
V(DISK_SMFR);
KERNEL_EXIT;

/*    DEC WRITER                                                    */

KERNEL_ENTRY;
V(DECW_PROCESS#);
KERNEL_EXIT;

/*    TTY                                                           */

KERNEL_ENTRY;
V(TTY_PROCESS#);
KERNEL_EXIT;

/*    SCOPE1                                                        */

KERNEL_ENTRY;
V(SCOPE1_PROCESS#);


KERNEL_EXIT;

/*    SCOPE2                                                        */

KERNEL_ENTRY;
V(SCOPE2_PROCESS#);
KERNEL_EXIT;
```

3.2.2 Parameter Checker (PCHECK)

The Parameter Checker CPC, PCHECK, is a kernel level internal SKCPP function that is called by only one other kernel level internal function, GATE. PCHECK calls all of the user level external functions as well as other kernel level internal functions. It is written in Project SUE System Language.

3.2.2.1 Description

PCHECK calls an externally reachable function if all input parameters are within acceptable ranges and if the seg# parameter (if required) identifies a segment in the process's WS.

It checks the function code: if FUNCTION_CODE_APARM is less than FUNCTION_CODE_MIN or greater than FUNCTION_CODE_MAX, PCHECK returns the SEVERE_FLAG. Otherwise, it sets FUNCTION_CODE equal to FUNCTION_CODE_APARM. It then performs a P on the kernel semaphore to block the kernel until after the function is performed, preventing more than one process from occupying the kernel at a time. PARM_FLAGS is assigned FUNCTION_ARRAY(FUNCTION_CODE), which indicates which parameters the function takes, and RC is set to OK_FLAG.

If the seg# parameter is required, PCHECK tests if the parameter supplied lies within the accepted range, as follows: if the logical and of PARM_FLAGS and SEG#_FLAG is not equal to zero, the specified function requires the seg# parameter. SEG#_PARM is set equal to SEG#_APARM. Then, if SEG#_PARM is less than SEG#_MIN or greater than SEG#_MAX, RC is set to SEVERE_FLAG.

PCHECK also checks that the seg# identifies a segment in the process's WS if the seg# parameter is required. If RC is not SEVERE_FLAG, ASTE#_PARM is assigned the value, PS_SEG(SEG#_PARM). Now, if the logical and of ASTE#_PARM and SEG_FLAG is not equal to zero, PS_SEG(SEG#_PARM) held a seg#. Therefore, it must have been on the free segment# chain. Hence, the segment specified was not in the WS of the process, so RC is set to zero. Otherwise, PS_SEG(SEG#_PARM) held an aste# and RC remains set to OK_FLAG.

In a similar manner, if the offset, classification, mode, reg#, and process# parameters are required, PCHECK checks that the values supplied lie within their acceptable ranges. Then, to CAT_PARM, SEG_TYPE_PARM, SIZE_PARM, USER_PARM, PROJECT_PARM, and MESSAGE_PARM, it assigns the respective values of CAT_APARM, SEG_TYPE_APARM logical and DIR_TYPE_MASK, SIZE_APARM, USER_APARM logical and ACL_USER_MASK, PROJECT_APARM, and MESSAGE_APARM.

If RC is not SEVERE_FLAG, PCHECK uses a case selector to call the function specified by FUNCTION_CODE and set RC to the value returned. To prevent compile-time parse stack overflow due to an implementation quirk, the case selection is broken up into two cases, FUNCTION_CODE between 1 and 9 and FUNCTION_CODE between 10 and 20, selected by an IF statement.

PCHECK then calls V to increment the kernel semaphore, and returns RC.

There are two short functions that are incorporated into the case selector under their FUNCTION_CODE tags. They are the T function whose FUNCTION_CODE tag is 12 and the PROCID (process identification) function whose FUNCTION_CODE tag is 18. Both have macros, in context NOFORN, associated with them.

The T function, which is user callable, is an inquiry about the semaphore count which does not require write access. If SMFR_COUNT (ASTE_PARM) is less than zero, T sets RC to ERR_FLAG.

The PROCID function merely assigns RC the value of PS_CURRENT_PROCESS. The process can use this to index the process directory directory and find its process directory or to index the I/O segment.

```
Function:  PCHECK
Parameters:  PCHECK(function_code,seg#,offset,class,cat,type,
             size,mode,user_id,project_id,reg#,process#,block#)
Effect:
IF (FUNCTION_CODE_MIN <= function_code <= FUNCTION_CODE_MAX &
   (not SEG#_PARM(function_code)|
   ((SEG#_MIN <= seg# <= SEG#_MAX) &
   PS_SEG_INUSE(TCP,seg#))) &
   (not OFFSET_PARM(function_code)|
   (OFFSET_MIN <= offset <= OFFSET_MAX)) &
   (not CLASS_PARM(function_code)|
   (CLASS_MIN <= class <= CLASS_MAX)) &
   (not CAT_PARM(function_code)|
   (cat ⊆ CATEGORY_SET)) &
   (not TYPE_PARM(function_code)|
   ((type = DIRECTORY) | (type = DATA))) &
   (not SIZE_PARM(function_code)|
   (SIZE_MIN <= size <= SIZE_MAX)) &
   (not MODE_PARM(function_code)|
   ((mode = WRITE) | (mode = READ) | (mode = NO))) &
   (not USER_ID_PARM(function_code)|
   (USER_ID_MIN <= user_id <= USER_ID_MAX)) &
   (not PROJECT_ID_PARM(function_code)|
   (PROJECT_ID_MIN <= project_id <= PROJECT_ID_MAX)) &
   (not REG#_PARM(function_code)|
   (REG#_MIN <= reg# <= REG#_MAX)) &
   (not PROCESS#_PARM(function_code)|
   (PROCESS#_MIN <= process# <= PROCESS#_MAX)) &
   (not BLOCK#_PARM(function_code)|
   (BLOCK#_MIN <= block # <= BLOCK#_MAX));
THEN:  Let aste# = PS_SEG(TCP,SEG#);
CASE of function_code:
   1.  CREATE(TCP,aste#,offset,class,cat,type,size);
   2.  DELETE(TCP,aste#,offset)
   3.  GIVE(TCP,aste#,offset,mode,user_id,project_id);
   4.  RESCIND(TCP,aste#,offset,user_id,project_id);
   5.  GETW(TCP,aste#,offset);
   6.  GETR(TCP,aste#,offset);
   7.  DCONNECT(seg#,aste#);
   8.  ENABLE(TCP,aste#,reg#);
   9.  DISABLE(reg#);
  10.  OUTERP(aste#);
  11.  OUTERV(aste#);
  12.  T(TCP,aste#);
  13.  IPCSEND(process#,message,USER_DOMAIN);
  14.  IPCRCV;
```

15. STARTP(TCP,user_id,project_id,class,cat,process#,offset);
16. STOPP;
17. CHANGEO(TCP,aste#,offset,class,cat);
18. KPROCID(TCP);
19. INITH(TCP,aste#,offset,cat);
20. READIR(TCP,aste#,offset);

ELSE: RC(TCP) = NO
END;

3.2.2.2 N/A

3.2.2.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
GATE	CREATE
	DELETE
	GIVE
	RESCIND
	OUTERP
	OUTERV
	STARTP
	STOPP
	CHANGEO
	INITH
	READIR
	IPCRCV
	IPCSEND
	GETW
	GETR
	ENABLE
	DISABLE
	DCONNECT
	P
	V

3.2.2.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function PCHECK. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1.

For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
PS_SEG	FUNCTION_CODE	SEG#_PARM
PS_CURRENT_PROCESS	SEG#	OFFSET_PARM
SEG#_APARM	OFFSET	CLASS_PARM
OFFSET_APARM	CLASS	CAT_PARM
CLASS_APARM	CAT	SEG_TYPE_PARM
CAT_APARM	TYPE	SIZE_PARM
SEG_TYPE_APARM	SIZE	MODE_PARM
SIZE_APARM	MODE	USER_PARM
MODE_APARM	USER_ID	PROJECT_PARM
USER_APARM	PROJECT_ID	REG#_PARM
PROJECT_APARM	REG#	PROCESS#_PARM
REG#_APARM	PROCESS#	MESSAGE_PARM
PROCESS#_APARM	BLOCK#	ASTE#_PARM
MESSAGE_APARM		PARM_FLAGS
FUNCTION_CODE_APARM		FUNCTION_CODE_TYPE
FUNCTION_CODE		RC
FUNCTION_ARRAY		

Constants

ACL_USER_MASK	OFFSET_MIN
CLASS_FLAG	OK_FLAG
CLASS_MAX	PROCESS#_FLAG
CLASS_MIN	PROCESS#_MAX
DIR_TYPE_MASK	PROCESS#_MIN
ERR_FLAG	READ$EXECUTE_ACCESS
FUNCTION_CODE_MAX	REG_FLAG
FUNCTION_CODE_MIN	REG#_MAX
KERNEL_SMFR	REG#_MIN
MODE_FLAG	SEG#_FLAG
NO_ACCESS	SEG#_MAX
OFFSET_FLAG	SEG#_MIN
OFFSET_MAX	SEVERE_FLAG
	WRITE$READ$EXECUTE_ACCESS

3.2.2.5 Limitations

PCHECK returns SEVERE_FLAG if the parameters passed to it do not fall within the acceptable ranges or if the seg# (if required) is not in the process's WS.

3.2.2.6 Listing

```
DATA PCHECK RETURNS (RC);


PROGRAM PCHECK;

     TYPE FUNCTION_CODE_TYPE = (0 TO FUNCTION_CODE_MAX);
     DECLARE
          FUNCTION_CODE_TYPE (FUNCTION_CODE);
     DECLARE
          WORD (SEG#_PARM, OFFSET_PARM, CLASS_PARM, CAT_PARM, SEG_TYPE_PARM, SIZE_PARM,
               MODE_PARM, USER_PARM, PROJECT_PARM, REG#_PARM, PROCESS#_PARM,
               MESSAGE_PARM, ASTE#_PARM, PARM_FLAGS);

     /*    CHECK FUNCTION CODE                                                        */

     IF FUNCTION_CODE_APARM < FUNCTION_CODE_MIN;
       THEN:
....      RETURN WITH SEVERE_FLAG;
     END;

     IF FUNCTION_CODE_APARM > FUNCTION_CODE_MAX;
       THEN:
....      RETURN WITH SEVERE_FLAG;
     END;

     FUNCTION_CODE := FUNCTION_CODE_APARM;
     P(KERNEL_SMFR);
     PARM_FLAGS := FUNCTION_ARRAY(FUNCTION_CODE);
     RC := OK_FLAG;

     /*    CHECK SEG# PARAMETER IF REQUIRED                                           */

     IF (PARM_FLAGS & SEG#_FLAG) ¬= 0;
       THEN: SEG#_PARM := SEG#_APARM;

          IF SEG#_PARM < SEG#_MIN;
            THEN: RC := SEVERE_FLAG;
          END;

          IF SEG#_PARM > SEG#_MAX;
            THEN: RC := SEVERE_FLAG;
          END;

          IF RC ¬= SEVERE_FLAG;
            THEN: ASTE#_PARM := PS_SEG(SEG#_PARM);

               IF (ASTE#_PARM & SEG_FLAG) ¬= 0;
                 THEN: RC := SEVERE_FLAG;
               END;

          END;

     END;
```

```
/*    CHECK OFFSET PARM IF REQUIRED                                          */

IF (PARM_FLAGS & OFFSET_FLAG) ¬= 0;
  THEN: OFFSET_PARM := OFFSET_APARM;


     IF OFFSET_PARM < OFFSET_MIN;
       THEN: RC := SEVERE_FLAG;
     END;

     IF OFFSET_PARM > OFFSET_MAX;
       THEN: RC := SEVERE_FLAG;
     END;

END;

/*    CHECK CLASSIFICATION PARAMETER IF REQUIRED                             */

IF (PARM_FLAGS & CLASS_FLAG) ¬= 0;
  THEN: CLASS_PARM := CLASS_APARM;

     IF CLASS_PARM < CLASS_MIN;
       THEN: RC := SEVERE_FLAG;
     END;

     IF CLASS_PARM > CLASS_MAX;
       THEN: RC := SEVERE_FLAG;
     END;

END;

/*    NO SYNTACTIC ERROR CHECKING FOR CATEGORY PARAMETER                     */

CAT_PARM := CAT_APARM;

/*    NO CHECKING OF SEG_TYPE PARAMETER                                      */

SEG_TYPE_PARM := (SEG_TYPE_APARM & DIR_TYPE_MASK);

/*    NO CHECKING OF SIZE PARAMETER                                          */

SIZE_PARM := SIZE_APARM;

/*    CHECK MODE PARM IF REQUIRED                                            */

IF (PARM_FLAGS & MODE_FLAG) ¬= 0;
  THEN: MODE_PARM := MODE_APARM;

     IF (MODE_PARM ¬= WRITE$READ$EXECUTE_ACCESS) & (MODE_PARM ¬= READ$EXECUTE_ACCESS
          ) & (MODE_PARM ¬= NO_ACCESS);
       THEN: RC := SEVERE_FLAG;
     END;

END;

/*    NO CHECKING OF USER PARAMETER                                          */

USER_PARM := (USER_APARM & ACL_USER_MASK);

/*    NO CHECKING OF PROJECT PARAMETER                                       */
```

```
PROJECT_PARM := PROJECT_APARM;

/*    CHECK REG# IF REQUIRED                                                    */

IF (PARM_FLAGS & REG_FLAG) ¬= 0;
  THEN: REG#_PARM := REG#_APARM;

     IF REG#_PARM < REG#_MIN;
       THEN: RC := SEVERE_FLAG;
     END;

     IF REG#_PARM > REG#_MAX;
       THEN: RC := SEVERE_FLAG;
     END;

END;

/*    CHECK PROCESS# PARAMETER IF REQUIRED                                      */

IF (PARM_FLAGS & PROCESS#_FLAG) ¬= 0;
  THEN: PROCESS#_PARM := PROCESS#_APARM;

     IF PROCESS#_PARM < PROCESS#_MIN;
       THEN: RC := SEVERE_FLAG;
     END;

     IF PROCESS#_PARM > PROCESS#_MAX;
       THEN: RC := SEVERE_FLAG;
     END;

END;

/*    NO CHECKING OF MESSAGE PARAMETER                                          */

MESSAGE_PARM := MESSAGE_APARM;

IF RC ¬= SEVERE_FLAG;
  THEN:
     IF FUNCTION_CODE < 10;
       THEN:

          CASE FUNCTION_CODE_TYPE TAG FUNCTION_CODE;
            1: RC := CREATE(ASTE#_PARM, OFFSET_PARM, CLASS_PARM, CAT_PARM,
                    SEG_TYPE_PARM, SIZE_PARM);
            2: RC := DELETE(ASTE#_PARM, OFFSET_PARM);
            3: RC := GIVE(ASTE#_PARM, OFFSET_PARM, MODE_PARM, USER_PARM,
                    PROJECT_PARM);
            4: RC := RESCIND(ASTE#_PARM, OFFSET_PARM, USER_PARM, PROJECT_PARM);
            5: RC := GETW(ASTE#_PARM, OFFSET_PARM);
            6: RC := GETR(ASTE#_PARM, OFFSET_PARM);
            7: DCONNECT(SEG#_PARM, ASTE#_PARM);
            8: RC := ENABLE(ASTE#_PARM, REG#_PARM);
            9: DISABLE(REG#_PARM);
          END;
```

```
      ELSE:

         CASE FUNCTION_CODE_TYPE TAG FUNCTION_CODE;
           10: RC := OUTERP(ASTE#_PARM);
           11: RC := OUTERV(ASTE#_PARM);
           12:
              IF SMFR_COUNT(ASTE#_PARM)<= 0;
                THEN: RC := ERR_FLAG;
              END;

           13: IPCSEND(PROCESS#_PARM, MESSAGE_PARM, 0);
           14: RC := IPCRCV;
           15: RC := STARTP(USER_PARM, PROJECT_PARM, CLASS_PARM, CAT_PARM,
                      PROCESS#_PARM, OFFSET_PARM);
           16: STOPP;
           17: RC := CHANGEO(ASTE#_PARM, OFFSET_PARM, CLASS_PARM, CAT_PARM);
           18: RC := PS_CURRENT_PROCESS;
           19: RC := INITH(ASTE#_PARM, OFFSET_PARM, CAT_PARM);
           20: RC := READIR(ASTE#_PARM, OFFSET_PARM);
         END;

    END;

END;

V(KERNEL_SMFR);
```

3.2.3 Create Segment (CREATE)

The Create Segment CPC, CREATE, is a user level external SKCPP function that is called by user level external programs with the parameters seg#, offset, class, cat, type, and size. CREATE calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.3.1 Description

CREATE creates a segment if security and implementation requirements are met. It calls WRITEDIR which sets RC to OK_FLAG if the intended parent segment is a directory to which the process has write access; otherwise, WRITEDIR sets ERR_FLAG. CREATE then sets RC to ERR_FLAG if the classification of the new segment is less than that of the parent segment, if the category set of the new segment does not include that of the parent segment, if the size of the new segment is not allowable, or if the offset does not identify a free directory entry. If RC is not set to ERR_FLAG, DALLOC allocates disk space for the segment with address DISK_ADR. DIR_TYPE is assigned SEG_TYPE or DIR_UNINITIALIZED or CLASS, DIR_CAT is assigned CAT, DIR_DISK is assigned DISK_ADR, and DIR_SIZE is filled in with SIZE.

```
Function:  CREATE
Parameters:  CREATE(process#,aste#,offset,class,cat,type,
             size)
Effect:
IF not AST_WAL(aste#,process#) |
   (class < AST_CLASS(aste#)) |
   (cat ⊅ AST_CAT(aste#)) |
   (AST_TYPE(aste#) ≠ DIRECTORY) |
   ('DIR_SIZE'(aste#,offset) ≠ 0) |
   (size ∉ SIZE_SET) |
   ((type = DIRECTORY) & (size ≠ DIRECTORY_SIZE))
THEN:  RC(process#) = NO;
ELSE:  DIR_TYPE(aste#,offset) = type;
   DIR_STATUS(aste#,offset) = UNINITIALIZED;
   DIR_CLASS(aste#,offset) = class;
   DIR_CAT(aste#,offset) = cat;
   DIR_SIZE(aste#,offset) = size;
   DISK_ALLOC(size);
   DIR_DISK(aste#,offset) = NEXT_DISK_ADDRESS;
   DIR_ACL_HEAD(aste#,offset) = 0;
   RC(process#) = YES;
END;
```

3.2.3.2 N/A

3.2.3.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
PCHECK	WRITEDIR DALLOC

3.2.3.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function CREATE. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1 For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
AST_CLASS	ASTE#	DISK_ADR
AST_CAT	OFFSET	RC
DIR_SIZE	CLASS	
DIR_TYPE	CAT	
DIR_CAT	SEG_TYPE	
DIR_DISK	SIZE	
DIR_SIZE		

Constants

AST_CLASS_MASK
BMT_SIZE2_ADR
DIR_UNINITIALIZED
ERR_FLAG
SIZE2

3.2.3.5 Limitations

CREATE returns with ERR_FLAG if the intended parent segment is a directory to which the process does not have write access, if the classification of the new segment is less than that of the parent segment, if the category set of the new segment does not include that of the parent segment, if the size of the new segment is not allowable, or if the offset does not identify a free directory entry.

3.2.3.6 Listing

```
DATA CREATE(ASTE#, OFFSET, CLASS, CAT, SEG_TYPE, SIZE) RETURNS (RC);
     DECLARE
          PROCEDURE ACCEPTS (WORD) RETURNS (WORD) (DALLOC);



PROGRAM CREATE;
     DECLARE
          WORD (DISK_ADR);

     /*    SECURITY CHECKS FIRST                                           */

     /*    CREATE IS INTERPRETATIVE DIRECTORY WRITE                        */

     RC := WRITEDIR(ASTE#);

     /*    CHECK COMPATIBILITY REQUIREMENTS                                */

     IF CLASS < (AST_CLASS(ASTE#) & AST_CLASS_MASK);
       THEN: RC := ERR_FLAG;
     END;
```

```
IF CAT ¬= (AST_CAT(ASTE#) | CAT);
  THEN: RC := ERR_FLAG;
END;

/*    IMPLEMENTATION CHECKS                                          */

/*    CHECK SIZE PARAMETER - ONLY SIZE2 ALLOWED AT THIS TIME         */

IF SIZE ¬= SIZE2;
  THEN: RC := ERR_FLAG;
END;

/*    CHECK OFFSET PARAMETER                                         */

IF DIR_SIZE(OFFSET) ¬= 0;
  THEN: RC := ERR_FLAG;
END;

IF RC = ERR_FLAG;
  THEN:
....    RETURN;
END;

/*    ALLOCATE A DISK AREA FOR THE SEGMENT                           */

DISK_ADR := DALLOC(BMT_SIZE2_ADR);

/*    CHECKING COMPLETE - PERFORM STATE CHANGE                       */

/*    FILL IN DIRECTORY ENTRY                                        */

DIR_TYPE(OFFSET) := (SEG_TYPE | DIR_UNINITIALIZED | CLASS);
DIR_CAT(OFFSET) := CAT;
DIR_DISK(OFFSET) := DISK_ADR;
DIR_SIZE(OFFSET) := SIZE;
```

3.2.4 Delete (DELETE)

The Delete CPC, DELETE, is a user level external SKCPP function that is called by user level external programs with the parameters seg# and offset. DELETE calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.4.1 Description

DELETE removes all ACL elements from the parent directory entry identified by ASTE#, OFFSET, deactivates the specified segment, frees its disk space, and marks the parent directory entry available. If WRITEDIR returns ERR_FLAG indicating that the parent segment is not a directory to which the calling process has write access or if the entry to be deleted is of size zero, DELETE returns with ERR_FLAG. If not, DELETE calculates the logical and of DIR_TYPE and DIR_TYPE_MASK to separate type information from the status and class. If the result is not DIR_TYPE_DIRECTORY, the entry is a data segment; DELETSEG is called to remove all ACL elements, deactivate the seg- ment, free its disk space, and set DIR_SIZE = Ø which marks the

entry available. If the result equals DIR_TYPE_DIRECTORY, DELETSEG cannot be called until the directory is empty.

DELETE then begins the directory deletion cycle: SPASTE# and SOFFSET are assigned the values of the ASTE# and OFFSET of the original directory to be deleted.

The process to find and delete the lowest directory in the hierarchy starting with the specified directory then follows: DELETE finds the disk address of the directory specified by SPASTE#, SOFFSET. It then calls HASH, which returns the directory's own ASTE# if it is active, or Ø if it is not. If it is Ø, ACT is called to activate the directory and SASTE# is set to the ASTE# of the directory to be deleted. Its segment descriptors are then loaded by GETDIR. DELETE then scans the entries of the directory to be deleted starting with TOFFSET equal to OFFSET_MIN. If the size of the entry identified by TOFFSET is zero, the scan of directory entries is continued. Otherwise, DELETE tests if the DIR_TYPE of the segment and DIR_TYPE_MASK equals DIR_TYPE_DIRECTORY. If so, this entry is itself another directory whose entries must be deleted before it can be deleted. In this case, DELETE resets SPASTE# and SOFFSET to SASTE# and TOFFSET and repeats the directory deletion process. If not, the entry is a data segment which DELETSEG deletes. The entry scan continues until TOFFSET equals OFFSET_MAX. Then, the directory is empty and DELETE can call DELETSEG to remove it. If SPASTE# is not equal to ASTE#, the last directory deleted was part of the sub-hierarchy of the directory orginally specified to be deleted. DELETE loads the segment descriptors of the original directory and continues the directory deletion cycle. If SPASTE# = ASTE#, the last directory deleted was the original segment; DELETE sets RC to OK_FLAG and returns control to the calling program.

```
Function:  DELETE
Parameters:  DELETE(process#,aste#,offset)
Effect:
IF not AST_WAL(aste#,process#) |
   (AST_TYPE(aste#) ≠ DIRECTORY) |
   ('DIR_SIZE'(aste#,offset) = 0);
THEN:  RC(process#) = NO;
ELSE:
   IF 'DIR_ACL_HEAD'(aste#,offset) ≠ 0;
   THEN:  Let acle# = FINDEND(aste#,'DIR_ACL_HEAD'(aste#,
          offset);
          ACL_CHAIN(aste#,acle#) = 'ACL_CHAIN'(aste#,0);
          ACL_CHAIN(aste#,0) = 'DIR_ACL_HEAD'(aste#,offset);
   END;
```

```
      DIR_SIZE(aste#,offset) = 0;
      DELETSEG(aste#,offset)
      IF DIR_TYPE(aste#,offset) = DIRECTORY;
      THEN: DELETSEG(aste#,offset)
      END;
      RC(process#) = YES;
   END;
```

3.2.4.2 N/A

3.2.4.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
PCHECK	GETDIR
	DELETSEG
	WRITEDIR
	HASH
	ACT

3.2.4.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function DELETE. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
DIR_SIZE	ASTE#	SPASTE#
DIR_TYPE	OFFSET	SASTE#
		SOFFSET
		TOFFSET
		RC

Constants

DIR_TYPE
DIR_TYPE_DIRECTORY
DIR_TYPE_MASK
OK_FLAG

3.2.4.5 Limitations

DELETE returns ERR_FLAG if the parent segment is not a directory to which the calling process has access or if the entry to be deleted is of size zero. Otherwise, RC = OK_FLAG.

3.2.4.6 Listing

```
DATA DELETE(ASTE#, OFFSET) RETURNS (RC);
     DECLARE
          PROCEDURE ACCEPTS (WORD) (GETDIR),
          PROCEDURE ACCEPTS (WORD, WORD) (DELETSEG);



PROGRAM DELETE;
     DECLARE
          WORD (SPASTE#, SASTE#, SOFFSET, TOFFSET);

     /*    SECURITY CHECKS FIRST                                              */

     /*    DELETE IS AN INTERPRETATIVE DIRECTORY WRITE                        */

     IF WRITEDIR(ASTE#) = ERR_FLAG;
       THEN:
....      RETURN WITH ERR_FLAG;
     END;

     /*    IMPLEMENTATION CHECKS                                              */

     IF DIR_SIZE(OFFSET) = 0;
       THEN:
....      RETURN WITH ERR_FLAG;
     END;

     /*    ELIMINATE CASE OF DATA SEGMENT                                     */

     IF (DIR_TYPE(OFFSET) & DIR_TYPE_MASK) ¬= DIR_TYPE_DIRECTORY;
       THEN: DELETSEG(ASTE#, OFFSET);
       ELSE: <OUTER_CYCLE>
          CYCLE

               /*    START AT THE TOP OF THE CHAIN AND WORK DOWN              */

               SPASTE# := ASTE#;
               SOFFSET := OFFSET;
               <INNER_CYCLE>
               CYCLE

                    /*    ASSURE DIRECTORY TO BE DELETED IS ACTIVE            */

                    SASTE# := HASH(DIR_DISK(SOFFSET));

                    IF SASTE# = 0;
                      THEN: SASTE# := ACT(SPASTE#, SOFFSET);
                    END;

                    /*    LOAD SEGMENT DESCRIPTORS                            */

                    GETDIR(SASTE#);
```

```
/*    SEARCH THE DIRECTORY FOR ENTRIES OF NON-ZERO SIZE            */

DO TOFFSET := OFFSET_MIN TO OFFSET_MAX;

     IF DIR_SIZE(TOFFSET) ¬= 0;
       THEN:
          IF (DIR_TYPE(TOFFSET) & DIR_TYPE_MASK) = DIR_TYPE_DIRECTORY;
            THEN: /* GAIN ACCESS TO A DIRECTORY ENTRY--PREPARE TO
                    SEARCH */
               SPASTE# := SASTE#;
               SOFFSET := TOFFSET;
....           REPEAT <INNER_CYCLE>;
            ELSE: /* DELETE A DATA SEGMENT ENTRY */
               DELETSEG(SASTE#, TOFFSET);
          END;

     END;

END;

/*    THIS DIRECTORY IS EMPTY--DELETE IT                             */

GETDIR(SPASTE#);
DELETSEG(SPASTE#, SOFFSET);

/*    FINISHED?                                                      */

.... EXIT <OUTER_CYCLE> WHEN SPASTE# = ASTE#;

/*    LOAD SEGMENT DESCRIPTORS OF ORIGINAL DIRECTORY                 */

GETDIR(ASTE#);
.... REPEAT <OUTER_CYCLE>;
END <INNER_CYCLE>;

END <OUTER_CYCLE>;

END;

RC := OK_FLAG;
```

3.2.5 Give Access (GIVE)

The Give Access CPC, GIVE, is a user level external SKCPP function that is called by user level external programs with the parameters seg#, offset, mode, user, and project. GIVE calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.5.1 Description

GIVE adds an ACL element to the directory entry of a segment if all constraints are satisfied. It calls WRITEDIR which returns ERR_FLAG if the process does not have write access to the parent directory of the segment. If WRITEDIR returns ERR_FLAG or if DIR_DISK for the segment is zero indicating the segment is non-existent, GIVE returns with ERR_FLAG. GIVE then scans the ACL beginning at the element specified by DIR_ACL_HEAD until ACL_CHAIN equals zero.

If, for an existing element, the logical and of ACL_USER and ACL_USER_MASK matches the USER specified in the function call, and ACL_PROJECT matches the PROJECT specified in the function call, GIVE returns with ERR_FLAG. Otherwise, it allocates an ACL element ACLE# equal to ACL_CHAIN (Ø), which holds the head of the free element chain. If ACLE# equals zero, no more elements are free, and GIVE returns with ERR_FLAG; if not,GIVE resets ACL_CHAIN(Ø) to the number of the next element held in ACL_CHAIN(ACLE#). GIVE then finds the proper position for the new element. POSITION is set to zero. If DIR_ACL_HEAD is zero, the ACL is empty and the element may be inserted at POSITION zero. If the ACL is not empty, and USER is ALL_USERS and PROJECT is ALL_PROJECTS the new element belongs at the end of the list: GIVE finds this by setting POSITION to DIR_ACL_HEAD(OFFSET) and resetting it to ACL_CHAIN(POSITION) repeatedly until ACL_CHAIN(POSITION) equals zero. If USER is ALL_USERS or PROJECT is ALL_PROJECTS but not both, GIVE tests if the first entry in the ACL, ACL_USER(DIR_ACL_HEAD(OFFSET)) and ACL_USER_MASK is ALL_USERS. If so, the element can be placed first in the list at POSITION zero. If not, POSITION is set to DIR_ACL_HEAD(OFFSET), and until ACL_CHAIN(POSITION) is zero marking the end of the chain or ACL_USER(ACL_CHAIN(POSITION)) and ACL_USER_MASK is ALL_USERS, POSITION is reset to ACL_CHAIN(POSITION). GIVE now fills in the new element, ACL_USER(ACLE#) assigned USER or MODE and ACL_PROJECT(ACLE#) assigned PROJECT. If POSITION is zero, the element is entered at the beginning of the list by setting ACL_CHAIN(ACLE#) to DIR_ACL_HEAD (OFFSET) and DIR_ACL_HEAD(OFFSET) to ACLE#. Otherwise, the element is inserted after the ACL element identified by POSITION: ACL_CHAIN (ACLE#) is assigned ACL_CHAIN(POSITION) and ACL_CHAIN(POSITION) is assigned ACLE#. Finally, if MODE is less than WRITE$READ$ECECUTE_ACCESS, SOADD is called to remove the segment from the work spaces of processes whose access rights have been rescinded. RC is then set to OK_FLAG.

```
Function:  GIVE
Parameters:  GIVE(process#,aste#,offset,mode,user_id,project_id)
Effect:
IF not AST_WAL(aste#,process#)|
   (AST_TYPE(aste#) ≠ DIRECTORY)|
   (DIR_SIZE(aste#,offset) = 0)|
   DUPACL(aste#, 'DIR_ACL_HEAD'(aste#,offset), user_id,
   project_id)|
   ('ACL_CHAIN'(aste#, 0) = 0);
THEN:  RC(process#) = NO;
ELSE:  Let acle# = 'ACL_CHAIN'(aste#,0);
   ACL_CHAIN(aste#,0) = 'ACL_CHAIN'(aste#,acle#);
   Let position = FACLPOS(aste#, 'DIR_ACL_HEAD'(aste#,offset),
   user_id, project_id);
```

```
   IF position = 0;
   THEN:  ACL_CHAIN(aste#,acle#) =
      'DIR_ACL_HEAD'(aste#,offset);
      DIR_ACL_HEAD(aste#,offset) = acle#;
   ELSE:  ACL_CHAIN(aste#,acle#) = 'ACL_CHAIN'(aste#,position);
      ACL_CHAIN(aste#,position) = acle#;
   END;
   ACL_USER(aste#,acle#) = user_id;
   ACL_PROJECT(aste#,acle#) = project_id;
   ACL_MODE(aste#,acle#) = mode;
   SOADD(aste#,offset);
   RC(process#) = YES;
END;

Function:  DUPACL
Parameters:  DUPACL(aste#,acle#,user_id,project_id);
Value:
IF acle# = 0;
THEN:  FALSE;
ELSE:
   IF (ACL_USER(aste#,acle#) = user_id) &
      (ACL_PROJECT(aste#,acle#) = project_id);
   THEN:  TRUE;
   ELSE:  DUPACL(aste#,ACL_CHAIN(aste#,acle#),user_id,
      project_id);
   END;
END;

Function:  FACLPOS
Parameters:  FACLPOS(aste#,acle#,user_id,project_id)
Value:
IF acle# = 0;
THEN:  0;
ELSE:
   IF (user_id = ALL_USERS) &
      (project_id = ALL_PROJECTS);
   THEN:  FINDEND(aste#,acle#);
   ELSE:
      IF (user_id = ALL_USERS)
         (project_id = ALL_PROJECTS);
      THEN:
```

```
            IF ACL_USER(aste#,acle#) = ALL_USERS;
            THEN:  0;
            ELSE:  FINDUSER(aste#,acle#);
            END;
        ELSE:  0;
        END;
    END;
END;

Function:  FINDEND
Parameters:  FINDEND(aste#, acle#)
Value:
IF ACL_CHAIN(aste#, acle#) ≠ 0;
THEN:  FINDEND(aste#, ACLE_CHAIN(aste#, acle#));
ELSE:  acle#;
END;

Function:  FINDUSER
possible values:  acle#
Parameters:  FINDUSER(aste#, acle#);
Value:
IF (ACL_CHAIN(aste#, acle#) = 0)|
   (ACL_USER(aste#, acle#) = ALL_USERS
THEN:  acle#;
ELSE:  FINDUSER(aste#, ACL_CHAIN(aste#, acle#));
END;
```

3.2.5.2 N/A

3.2.5.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
PCHECK	WRITEDIR SOADD

3.2.5.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function GIVE. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
DIR_DISK	ASTE#	ACLE#
DIR_ACL_HEAD	OFFSET	INDEX
ACL_USER	MODE	POSITION
ACL_PROJECT	USER	RC
ACL_CHAIN	PROJECT	

Constants

ACL_USER_MASK
ALL_PROJECTS
ALL_USERS
ERR_FLAG
OK_FLAG
WRITE$READ$EXECUTE_ACCESS

3.2.5.5 Limitations

GIVE returns ERR_FLAG if DIR_DISK for the segment is zero, if ACL_USER and ACL_USER_MASK matches the USER specified and ACL_PROJECT matches the PROJECT specified, or if no elements are free. Otherwise, RC = OK_FLAG.

3.2.5.6 Listing

```
DATA GIVE(ASTE#, OFFSET, MODE, USER, PROJECT) RETURNS (RC);


PROGRAM GIVE;
     DECLARE
          WORD (ACLE#, INDEX, POSITION);

     /*    SECURITY CHECKS FIRST                                          */

     /*    GIVE WRITES INTO DIRECTORY (INTREPRETATIVE WRITE)              */

     IF WRITEDIR(ASTE#) = ERR_FLAG;
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     /*    IMPLEMENTATION CHECKS                                          */

     /*    SEGMENT MUST EXIST                                             */

     IF DIR_DISK(OFFSET) = 0;
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

                                                                          */
```

```
/*     SEARCH FOR DUPLICATE ACL ELEMENT

     INDEX := DIR_ACL_HEAD(OFFSET);

     CYCLE
     .... EXIT WHEN INDEX = 0;

          IF ((USER = (ACL_USER(INDEX) & ACL_USER_MASK)) & (PROJECT = ACL_PROJECT(INDEX))
               );
            THEN:
....            RETURN WITH ERR_FLAG;
          END;

          INDEX := ACL_CHAIN(INDEX);
     END;

/*     ALLOCATE AN ACL ELEMENT                                                   */

     ACLE# := ACL_CHAIN(0);

     IF ACLE# = 0;
       THEN:
....      RETURN WITH ERR_FLAG;
     END;

/*     CHECKING COMPLETE - PERFORM STATE CHANGE                                  */

     ACL_CHAIN(0) := ACL_CHAIN(ACLE#);

/*     DETERMINE CORRECT POSITION FOR NEW ACL ELEMENT                            */

     POSITION := 0;

     IF DIR_ACL_HEAD(OFFSET) ¬= 0;
       THEN:
          IF ((USER = ALL_USERS) & (PROJECT = ALL_PROJECTS));
            THEN: POSITION := DIR_ACL_HEAD(OFFSET);

               CYCLE
               .... EXIT WHEN ACL_CHAIN(POSITION) = 0;
                    POSITION := ACL_CHAIN(POSITION);
               END;

            ELSE:
               IF ((USER = ALL_USERS) | (PROJECT = ALL_PROJECTS));
                 THEN:
                    IF (ACL_USER(DIR_ACL_HEAD(OFFSET)) & ACL_USER_MASK) ¬= ALL_USERS;
                      THEN: POSITION := DIR_ACL_HEAD(OFFSET);

                         CYCLE
                         .... EXIT WHEN (ACL_CHAIN(POSITION) = 0) | ((ACL_USER(ACL_CHAIN(
                                   POSITION)) & ACL_USER_MASK) = ALL_USERS);
                              POSITION := ACL_CHAIN(POSITION);
                         END;

                    END;

               END;

          END;

     END;
```

```
/*    FILL IN NEW ACL ELEMENT AND ADD TO CHAIN                                          */

ACL_USER(ACLE#)  :=  (USER | MODE);
ACL_PROJECT(ACLE#)  :=  PROJECT;

IF POSITION = 0;
  THEN: ACL_CHAIN(ACLE#)  := DIR_ACL_HEAD(OFFSET);
      DIR_ACL_HEAD(OFFSET)  := ACLE#;
  ELSE: ACL_CHAIN(ACLE#)  := ACL_CHAIN(POSITION);
      ACL_CHAIN(POSITION)  := ACLE#;
END;

/*    "GIVE" MAY RESCIND ACCESS RIGHTS                                                  */

IF MODE ¬= WRITE$READ$EXECUTE_ACCESS;
  THEN: SOADD(ASTE#, OFFSET);
END;

RC := OK_FLAG;
```

3.2.6 Rescind Access (RESCIND)

The Rescind Access CPC, RESCIND, is a user level SKCPP function that is called by user level external programs with the parameters seg#, offset, user, and project. RESCIND calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.6.1 Description

RESCIND moves an element from an entry's ACL to the directory's free element chain. If WRITEDIR shows that the ASTE# supplied does not identify a directory to which the process has write access, RESCIND returns ERR_FLAG. Otherwise, RESCIND searches the ACL for an element such that ACL_USER(INDEX) and ACL_USER_MASK equals USER and ACL_PROJECT equals PROJECT, starting with INDEX set to DIR_ACL_HEAD(OFFSET). If INDEX is Ø, the end of the list has been reached without finding the specified element, so ERR_FLAG is returned. If the element identified by INDEX is the one specified, RESCIND stops its search; otherwise SAVE_LAST is set to INDEX, INDEX is reset to ACL_CHAIN(INDEX), and the search is continued.

After the element is found, if INDEX equals DIR_ACL_HEAD(OFFSET) DIR_ACL_HEAD(OFFSET) is reset to ACL_CHAIN(INDEX). If not, ACL_CHAIN(SAVE_LAST) is set to ACL_CHAIN(INDEX). The element thus removed is then put at the head of the free chain by assigning ACL_CHAIN(INDEX) the value of ACL_CHAIN(Ø) and ACL_CHAIN(Ø) the value of INDEX. SOADD is called to remove the entry from the work spaces of processes whose access rights have been limited by RESCIND. RESCIND then returns with RC set to OK_FLAG.

```
Function:  RESCIND
Parameters:  RESCIND(process#, aste#, offset, user_id,
             project_id)
Effect:
IF not AST_WAL(aste#, process#)
   (AST_TYPE(aste#) ≠ DIRECTORY)
   (DIR_SIZE(aste#, offset) = 0)
   not DUPACL(aste#, 'DIR_ACL_HEAD' (aste#, offset), user_id,
   project_id);
THEN:  RC(process#) = NO;
ELSE:  Let acle# = FINDACLE(aste#, 'DIR_ACL_HEAD'(aste#,
       offset), user_id, project_id;
   IF  acle# = 'DIR_ACL_HEAD'(aste#, offset);
   THEN:  DIR_ACLE_HEAD(aste#, offset)
          ACL_CHAIN(aste#, acle#);
   ELSE:  Let pacle# = FINDPLACE(acle#);
          'DIR_ACL_HEAD'(aste#, offset), acle#);
          ACL_CHAIN(aste#, pacle#) = 'ACL_CHAIN'(aste#, acle#);
END;

Function: FINDACLE
Parameters:  FINDACLE(aste#, acle#, user_id, project_id)
Value:
IF  (ACL_USER(aste#, acle#) = user_id)&
    (ACL_PROJECT(aste, acle#) = project_id);
THEN:  acle#;
ELSE:  FINDACLE(aste#, ACL_CHAIN(aste#, acle#), user_id,
       project_id);
END:

Function:  FINDPACLE
Parameters:  FINDPACLE(aste#, vacle#, acle#)
Value:
IF  ACL_CHAIN(aste, vacle#) = acle#;
THEN:  vacle#;
ELSE:  FINDPACLE(aste#, ACL_CHAIN(aste#, vacle#,), acle#);
END:
```

3.2.6.2 N/A

3.2.6.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
PCHECK	WRITEDIR
	SOADD

3.2.6.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function RESCIND. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
DIR_ACL_HEAD	ASTE#	INDEX
ACL_USER	OFFSET	SAVE_LAST
ACL_PROJECT	USER	RC
ACL_CHAIN	PROJECT	

Constants

ALL_USER_MASK
ERR_FLAG
OF_FLAG

3.2.6.5 Limitations

RESCIND returns ERR_FLAG if the ASTE# supplied does not identify a directory to which the process has write access or if ACL_USER and ACL_USER_MASK does not match the USER specified and ACL_PROJECT does not match the PROJECT specified. Otherwise, RC = OK_FLAG.

3.2.6.6 Listing

```
DATA RESCIND(ASTE#, OFFSET, USER, PROJECT) RETURNS (RC);


PROGRAM RESCIND;
     DECLARE
          WORD (INDEX, SAVE_LAST);

     /*    SECURITY CHECKS FIRST                                                  */

     /*    RESCIND IS INTERPRETATIVE DIRECTORY WRITE                              */

     IF WRITEDIR(ASTE#) = ERR_FLAG;
       THEN:
....      RETURN WITH ERR_FLAG;
     END;
```

```
/*    IMPLEMENTATION CHECKS                                              */

/*    SEARCH FOR SPECIFIED ACL ELEMENT                                   */

INDEX := DIR_ACL_HEAD(OFFSET);

CYCLE

     IF INDEX = 0;
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     IF ((USER = (ACL_USER(INDEX) & ACL_USER_MASK)) & (PROJECT = ACL_PROJECT(INDEX))
          );
       THEN:
....        EXIT;
     END;

     SAVE_LAST := INDEX;
     INDEX := ACL_CHAIN(INDEX);
END;

/*    CHECKING COMPLETE - PERFORM STATE CHANGE                           */

/*    REMOVE FOUND ACL ELEMENT FROM CHAIN                                */

IF INDEX = DIR_ACL_HEAD(OFFSET);
  THEN: DIR_ACL_HEAD(OFFSET) := ACL_CHAIN(INDEX);
  ELSE: ACL_CHAIN(SAVE_LAST) := ACL_CHAIN(INDEX);
END;

/*    AND PUT ON FREE CHAIN                                              */

ACL_CHAIN(INDEX) := ACL_CHAIN(0);
ACL_CHAIN(0) := INDEX;

/*    NOW FOR THE MESSY PART                                             */

SOADD(ASTE#, OFFSET); /* SEARCH OUT AND DESTROY DESCRIPTORS */
RC := OK_FLAG;
```

3.2.7 Get Write Access (GETW)

The Get Write Access CPC, GETW, is a user level SKCPP function that is called by one other user level external function and by user level external programs with the parameters seg# and offset. GETW calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.7.1 Description

GETW gets write access to a segment if security requirements are met. It calls DSEARCH which returns ERR_FLAG if the user does not have WRITE$READ$EXECUTE_ACCESS according to the ACL. If DSEARCH returns ERR_FLAG, GETW returns with ERR_FLAG.

If PS_CURRENT_PROCESS does not equal EXEC_PROCESS#, the process is not trusted and the *- property must be enforced. The *-property requires that all objects to which a subject has write access have

the same security level and that all objects to which it has read access have a security level less than or equal to the write security level. Thus, if PS_CUR_CLASS is not identical to DIR_CLASS)OFFSET) and DIR_CLASS_MASK or if PS_CUR_CAT is not identical to DIR_CAT(OFFSET) GETW returns with ERR_FLAG.

If PS_CURRENT_PROCESS does not equal EXEC_PROCESS#, the process is trusted and only the preservation of security need be enforced. If PS_CUR_CLASS is less than DIR_CLASS(OFFSET) and DIR_CLASS MASK or if PS_CUR_CAT is not equal to DIR_CAT(OFFSET) or PS_CUR_CAT, GETW returns with ERR_FLAG.

Security checking complete, GETW calls CONNECT. If the process is not already connected to the seg and has a free seg#, the segment is activated if necessary and the process is added to the CPL and the WAL of the ASTE. GETW returns the seg# with which the process can identify the segment.

```
Function:  GETW
Parameters:  GETW(process#, aste#, offset)
Effect:
IF  (ASTE_TYPE(aste# ≠ DIRECTORY)|
    (DIR_SIZE(aste#, offset) = 0)|
    not DSEARCH(process#, aste#, DIR_ACL_HEAD(aste#, offset),
    WRITE);
THEN:  RC(process#) = NO;
ELSE:
   IF  PS_TYPE(process#) = TRUSTED;
       THEN:
          IF  (PS_CUR_CLASS(process#),< DIR_CLASS(aste#, offset))|
              (PS_CUR_CAT(process#) ⊉ DIR_CAT(aste#, offset));
              THEN:  RC(process#) = NO;
              ELSE:  CONNECT(process#, aste#, offset, WRITE);
          END;
       ELSE:
          IF  (PS_CUR_CLASS(process#) ≠ DIR_CLASS(aste#, offset))|
              (PS_CUR_CAT(process#) ≠ DIR_CAT(aste#, offset));
              THEN:  RC(process#) = NO;
              ELSE:  CONNECT(process#, aste#, offset, WRITE);
           END;
   END;
END;
```

3.2.7.2 N/A

3.2.7.3 Interfaces

Called By	Calls
STARTP	DSEARCH
PCHECK	CONNECT

3.2.7.4 Data Organization

Listed below are Security Kernel data base references and constants used by function GETW. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraps 3.3.1. For constants refer to Table I, List of constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
PS_CURRENT_PROCESS	ASTE#	RC
PS_CUR_CLASS	OFFSET	
PS_CUR_CAT		
DIR_CLASS		
DIR_CAT		

Constants

DIR_CLASS_MASK
ERR_FLAG
EXEC_PROCESS#
WRITE$READ$EXECUTE_ACCESS

3.2.7.5 Limitations

GETW returns ERR_FLAG if the user does not have WRITE$READ$ EXECUTE_ACCESS to the segment, if PS_CURRENT_PROCESS does not equal EXEC_PROCESS# or if PS_CUR_CLASS and PS_CUR_CAT does not match DIR_CLASS and DIR_CAT. Otherwise, RC = seg#.

3.2.7.6 Listing

```
DATA GETW(ASTE#, OFFSET) RETURNS (RC);
```

```
PROGRAM GETW;

     /*     SECURITY CHECKS FIRST                                            */

     /*     SEARCH DIRECTORY ACL                                             */

     IF DSEARCH(ASTE#, OFFSET, WRITE$READ$EXECUTE_ACCESS) = ERR_FLAG;
       THEN:
....       RETURN WITH ERR_FLAG;
     END;

     IF PS_CURRENT_PROCESS ¬= EXEC_PROCESS#;
       THEN:

          /*     CHECK FOR PRESERVATION OF SECURITY AND * -PROPERTY          */

          IF PS_CUR_CLASS ¬= (DIR_CLASS(OFFSET) & DIR_CLASS_MASK);
            THEN:
....            RETURN WITH ERR_FLAG;
          END;

          IF PS_CUR_CAT ¬= DIR_CAT(OFFSET);
            THEN:
....            RETURN WITH ERR_FLAG;
          END;

       ELSE:
          IF PS_CUR_CLASS < (DIR_CLASS(OFFSET) & DIR_CLASS_MASK);
            THEN:
....            RETURN WITH ERR_FLAG;
          END;

          IF PS_CUR_CAT ¬= (DIR_CAT(OFFSET) | PS_CUR_CAT);
            THEN:
....            RETURN WITH ERR_FLAG;
          END;

     END;

     /*     IMPLEMENTATION CHECKS                                            */

     /*     CONNECT PROCESS TO AST ENTRY FOR THIS SEGMENT                    */

     RC := CONNECT(ASTE#, OFFSET, WRITE$READ$EXECUTE_ACCESS);
```

3.2.8 Get Read Access (GETR)

The Get Read Access CPC, GETR, is a user level external SKCPP function that is called by one other user level external function and by user level external programs with the parameters seg# and offset. GETR calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.8.1 Description

GETR gets access to a segment identified by aste#, offset for a process, if security requirements are satisfied. It calls DSEARCH which scans the ACL of the segment and returns ERR_FLAG if the process lacks READ$EXECUTE_ACCESS. If DSEARCH returns

ERR_FLAG, GETR returns with ERR_FLAG. GETR then checks the preservation of security. If PS_CUR_CLASS is less than DIR_CLASS(OFFSET) and DIR_CLASS_MASK or or if PS_CUR_CAT does not equal DIR_CAT(OFFSET) or PS_CUR_CAT, ERR_FLAG is returned.

Otherwise, security checking is complete, and CONNECT is called. It adds the process to the CPL of the segment's ASTE and returns its seg# if the process is not already connected to the segment and has a free seg#. GETR then returns with the seg# which the process can subsequently use to refer to the segment.

```
Function:  GETR
Parameters:  GETR(process#, aste#, offset)
Effect:
IF  (AST_TYPE(aste#)  = DIRECTORY)|
    (DIR_SIZE(aste#, offset) = 0)|
    not DSEARCH(process#, aste#, DIR_ACL_HEAD(aste#, offset),
    READ)|
    (PS_CUR_CLASS(process#)< DIR_CLASS(aste#, offset))|
    (PS_CUR_CAT(process#) ⊉ DIR_CAT(aste#, offset));
    THEN:  RC(process#) = NO;
    ELSE:  CONNECT(process#, aste, offset, READ);
END;
```

3.2.8.2 N/A

3.2.8.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
STARTP	DSEARCH
PCHECK	CONNECT

3.2.8.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function GETR. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
PS_CUR_CLASS	ASTE#	RC
PS_CUR_CAT	OFFSET	

Global References | Function Parameters | Local References

DIR_CLASS
DIR_CAT

Constants

DIR_CLASS_MASK
ERR_FLAG
READ$EXECUTE_ACCESS

3.2.8.5 Limitations

GETR returns ERR_FLAG if the process does not have READ$EXECUTE_ACCESS to the segment or if PS_CUR_CLASS and PS_CUR_CAT do not match DIR_CLASS and DIR_CAT. Otherwise, RC = seg#.

3.2.8.6 Listing

```
DATA GETR(ASTE#, OFFSET) RETURNS (RC);

PROGRAM GETR;

     /*    SECURITY CHECKS FIRST                                          */

     /*    SEARCH DIRECTORY ACL                                           */

     IF DSEARCH(ASTE#, OFFSET, READ$EXECUTE_ACCESS) = ERR_FLAG;
       THEN:
....       RETURN WITH ERR_FLAG;
     END;

     /*    CHECK FOR PRESERVATION OF SECURITY AND * -PROPERTY             */

     IF PS_CUR_CLASS < (DIR_CLASS(OFFSET) & DIR_CLASS_MASK);
       THEN:
....       RETURN WITH ERR_FLAG;
     END;

     IF PS_CUR_CAT ¬= (DIR_CAT(OFFSET) | PS_CUR_CAT);
       THEN:
....       RETURN WITH ERR_FLAG;
     END;

     /*    IMPLEMENTATION CHECKS                                          */

     /*    CONNECT PROCESS TO AST ENTRY FOR THIS SEGMENT                  */

     RC := CONNECT(ASTE#, OFFSET, READ$EXECUTE_ACCESS);
```

3.2.9 Release Segment (DCONNECT)

The Release Segment CPC, DCONNECT, is a user level external SKCPP function that is called by user level external functions, one kernel level internal function, and by user external programs with

the parameter seg#. DCONNECT calls only one kernel level internal function. This function is unique in that it is known by two names: external to the Security Kernel by RELEASE and internal to the Security Kernel by DCONNECT. The rationale being the nomenclature RELEASE matches a function in the mathematical model (reference 2.1a) where as the procedure, while being true to the mathematical model, is the logical inverse of the internal procedure CONNECT. DCONNECT is written in Project SUE Language.

3.2.9.1 Description

DCONNECT releases a segment from the process's WS as long as the seg# is valid. It sets BLOCK# to AST_ADR(ASTE#), which holds the main memory address of the segment, if any, or zero. If BLOCK# is not zero and AST_UNLOCK(ASTE#) and AST_UNLOCK_MASK does not equal AST_UNLOCK_FLAG, the segment is in the AS of the process and must be disabled. In this case, starting with REG# set to zero until REG# equals REG#_MAX, DCONNECT tests if PS_SAR(REG#) equals BLOCK#. If so, DISABLE can be called to remove the segment identified by REG# from the AS.

The process can now be disconnected from the segment. AST_CPL (ASTE#) is assigned the logical and of PS_PROCESS_NOTMASK, which consists of all one's except for the bit corresponding to the process#, and AST_CPL(ASTE#). Similarly, AST_WAL(ASTE#) is assigned PS_PROCESS_NOTMASK and AST_WAL(ASTE#).

If the WAL is empty DCONNECT must free any processes that are blocked on the segment semaphore, since P and V require write access to the segment. Thus, if AST_WAL(ASTE#) equals zero and AST_CPL(ASTE#) and WIRED_DOWN_MASK does not equal WIRED_DOWN, DCONNECT calls V repeatedly while SMFR_COUNT is negative. On exiting this loop, SMFR_COUNT(ASTE#) is set to 1 and SMFR_POINTER (ASTE#) is set to Ø.

If AST_CPL(ASTE#) = Ø, the segment is not in the working space of any process. Therefore, DCONNECT sets AST_AGE_CHAIN(ASTE#) to AST_AGE_CHAIN(Ø) and AST_AGE_CHAIN(Ø) to ASTE#.

Finally, DCONNECT puts SEG# on the chain of free segment numbers; PS_SEG(SEG#) is set to PS_SEG(Ø) and PS_SEG(Ø) is reset to SEG# or SEG_FLAG.

```
Function:  RELEASE
Parameters:  RELEASE(process#, aste#, seg#)
Effect:
Let block# = 'AST_ADR'(aste#);
IF  (block# ≠ 0);
THEN:
   (∀reg#)  (REG#_MIN ≤ reg# ≤ REG_MAX) &
   IF  ('PS_SAR'(process#, reg#) = block#);
   THEN:  DISABLE(process#, reg#);
   END;
   END;
END;
AST_CPL(aste#, process#) = FALSE;
AST_WAL(aste#, process#) = FALSE;
IF not (∃i)(PROCESS#_MIX ≤ i ≤ PROCESS#_MAX) &
       (AST_CPL(aste#, i) = TRUE));
THEN:  AST(aste#);
END;
PS_SEG(process#, seg#) = 'PS_SEG'(process#, 0);
PS_SEG(process#, 0) = seg#;
PS_SEG_INUSE(process#, seg#) = FALSE;
```

3.2.9.2 N/A

3.2.9.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
PCHECK	DISABLE
STARTP	
STOPP	
SOADD	

3.2.9.4. Data Organization

Listed below are Security Kernel data base references and constants used by the function DCONNECT. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
PS_SAR	SEG#	BLOCK#
PS_SEG	ASTE#	REG#
AST_ADR		
AST_UNLOCK		
AST_CPL		
AST_WAL		
AST_AGE_CHAIN		
SMFR_COUNT		
SMFR_POINTER		

Constants

ASL
AST_UNLOCK_FLAG
BLOCKED
REG#_MAX
SEG_FLAG
WIRED_DOWN
WIRED_DOWN_MASK

3.2.9.5 Limitations

None

3.2.9.6 Listing

```
DATA DCONNECT(SEG#, ASTE#);

PROGRAM DCONNECT;
     DECLARE
          WORD (BLOCK#, REG#);

     /*    PROCESS MUST NOT HAVE ANY DESCRIPTORS ON SEGMENT LOCKED IN              */

     BLOCK# := AST_ADR(ASTE#);

     IF (BLOCK# ¬= 0) & ((AST_UNLOCK(ASTE#) & AST_UNLOCK_MASK) ¬= AST_UNLOCK_FLAG);
       THEN:

          /*    MY BLOCKS TO THEIRS                                                 */

          INLINE(ASL, BLOCK#);
          INLINE(ASL, BLOCK#);

          DO REG# := 0 TO REG#_MAX;

               IF PS_SAR(REG#) = BLOCK#;
                 THEN: DISABLE(REG#);
               END;

          END;

     END;
```

```
/*    DISCONNECT THIS PROCESS                                                    */

AST_CPL(ASTE#) := (AST_CPL(ASTE#) & PS_PROCESS_NOTMASK);
AST_WAL(ASTE#) := (AST_WAL(ASTE#) & PS_PROCESS_NOTMASK);

/*    RESET SEMAPHORES FOR SEGMENT WITH EMPTY WRITE ACCESS LIST                  */

IF (AST_WAL(ASTE#) = 0) & ((AST_CPL(ASTE#) & WIRED_DOWN_MASK) ¬= WIRED_DOWN);
  THEN: /* FREE PROCESSES BLOCKED ON THE SEGMENT SEMAPHORE */

     CYCLE
     .... EXIT WHEN SMFR_COUNT(ASTE#) >= 0;
          V(ASTE#);
     END;

     SMFR_COUNT(ASTE#) := 1;
     SMFR_POINTER(ASTE#) := 0;
END;

/*    AGE IF THIS IS THE LAST PROCESS TO DISCONNECT                              */

IF AST_CPL(ASTE#) = 0;
  THEN: AST_AGE_CHAIN(ASTE#) := AST_AGE_CHAIN(0);
     AST_AGE_CHAIN(0) := ASTE#;
END;

/*    PUT SEG# ON FREE CHAIN                                                     */

PS_SEG(SEG#) := PS_SEG(0);
```

3.2.10 Enable (ENABLE)

The Enable CPC, ENABLE, is a user level external SKCPP function that is called by other user level external functions and by user level external programs with the parameters seg# and reg#. ENABLE calls only kernel level internal functions. It is written in Project SUE System Language, including the Inline feature.

3.2.10.1 Description

ENABLE moves a segment from a process's WS to its AS if implementation constraints are satisfied. If REG# is greater than CROSS_REG#, it sets P_REG# to REG# + REG_CONSTANT; otherwise it sets P_REG# to REG#.

If PS_SAR(REG#) is not ∅, the register specified is not free and ENABLE returns with ERR_FLAG. It next tests whether sufficient space in the user's memory is available. If the logical and of AST_CPL(ASTE#) and WIRED_DOWN_MASK does not equal WIRED_DOWN and AST_SIZE(ASTE#) is greater than PS_MEM_QUOTA, ENABLE returns with ERR_FLAG. If not, implementation checks are complete.

ENABLE then insures that the segment is in main memory by calling SWAPIN if AST_ADR(ASTE#) is zero. If AST_UNLOCK(ASTE#) and AST_UNLOCK_MASK equals AST_UNLOCK_FLAG, the segment is currently eligible to be swapped out of main memory and must be removed from

the swap chain. Starting with INDWX set to Ø, ENABLE repeatedly sets NEXT to AST_SWAP_CHAIN(INDEX) and INDEX to NEXT, leaving the loop upon setting NEXT to ASTE#. It then assigns to AST_SWAP_CHAIN_(INDEX) the value of AST_SWAP_CHAIN(ASTE#), to AST_DES_COUNT(ASTE#) the value zero, and to AST_UNLOCK(ASTE#) the logical and of AST_UNLOCK(ASTE#) and AST_LOCK_MASK.

To determine the mode of access, ENABLE calculates the logical and of AST_TYPE(ASTE#) and AST_TYPE_MASK. If this equals AST_TYPE_DIRECTORY MODE is set to zero. Otherwise, if AST_WAL_(ASTE#) and P _PROCESS_MASK equals zero MODE is set to SDR_READ_ACCESS; if not, MODE is set equal to SDR_WRITE_ACCESS.

Sets priority level high through the Inline feature, and LSD is called to construct the descriptors and store them in the appropriate segmentation registers. If PS_CURRENT_PROCESS is THE_CURRENT_PROCESS, ENABLE loads the hardware registers as well: SDR(P_REG#) is assigned PS_SDR(REG#) and SAR(P_REG#) is assigned PS_SAR(REG#).

To complete the operation, sets priority level low through the Inline feature, and AST_DES_COUNT(ASTE#) is incremented if AST_DES_COUNT(ASTE#) is incremented. If AST_CPL(ASTE#) and WIRED_DOWN_MASK does not equal WIRED_DOWN, the user's memory space, PS_MEM_QUOTA is diminished by AST_SIZE(ASTE#). ENABLE then returns RC set equal to OK_FLAG.

```
Function:  ENABLE
Parameters:  ENABLE(process#, reg#)
Effect:
Let size = AST_SIZE(aste#);
IF  (PS_SAR(process#, reg#) ≠ 0)|
    ((AST_WIRED_DOWN(aste#) = OFF) &
    (size >'PS_MEM_QUOTE' (process#));
THEN:  RE(process#) = NO;
ELSE:
   IF  'AST_ADR'(ASTE#) = 0;
   THEN:  SWAPIN(aste#);
END;
IF  'AST_LOCK'(aste#) = UNLOCKED;
THEN:  LOCK(aste#);
END;
```

```
IF  AST_TYPE(aste#) = DIRECTORY
THEN:  Let mode = NO;
ELSE:
   IF  AST_WAL(aste#, process#) = TRUE;
   THEN:  Let mode = WRITE;
   ELSE:  Let mode = READ;
   END:
END;
LSD(AST_ADR(aste#), reg#, mode);
IF  AST_WIRED_DOWN(aste#) = OFF;
THEN:  PS_MEM_QUOTA(process#) = 'PS_MEM_QUOTA'(process#) -
       size;
END;
AST_DES_COUNT(aste#) = 'AST_DES_COUNT(aste#) + 1;
RC(process#) = YES
```

3.2.10.2 N/A

3.2.10/3 Interfaces

Refers to Figure 6, Function Call Matrix

Called By	Calls
PCHECK	SWAPIN
STARTP	LSD

3.2.10.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function ENABLE. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
PS_SAR	ASTE#	P_REG#
PS_SDR	REG#	MODE
PS_MEM_QUOTA		REG_ADR
PS_PROCESS_MASK		INDEX
PS_CURRENT_PROCESS		NEXT
AST_ADR		RC
AST_UNLOCK		
AST_SWAP_CHAIN		
AST_DES_COUNT		

Global References	Function Parameters	Local References
AST_TYPE		
SDR		
SAR		
THE_CURRENT_PROCESS		

Constants

AST_LOCK_MASK	PS_SDR_ADR
AST_TYPE_DIRECTORY	REG_CONSTANT
AST_TYPE_MASK	SDR_READ_ACCESS
AST_UNLOCK_FLAG	SDR_WRITE_ACCESS
AST_UNLOCK_MASK	SPLHIGH
CROSS_REG#	SPLLOW
ERR_FLAG	WIRED_DOWN
OK_FLAG	WIRED_DOWN_MASK

3.2.10.5 Limitations

ENABLE returns ERR_FLAG if PS_SAR(REG#) is not Ø, if AST_CPL(ASTE#) and WIRED_DOWN_MASK does not equal WIRED_DOWN and if AST_SIZE(ASTE) is greater than PS_MEM_QUOTA. Otherwise, RC=OK_FLAG.

3.2.10.6 Listing

```
DATA ENABLE(ASTE#, REG#) RETURNS (RC);




PROGRAM ENABLE;
     DECLARE
          WORD (P_REG#, MODE, REG_ADR, INDEX, NEXT);

     IF REG# > CROSS_REG#;
       THEN: P_REG# := REG# + REG_CONSTANT;
       ELSE: P_REG# := REG#;
     END;

     /*    REGISTER MUST BE FREE                                        */

     IF PS_SAR(REG#) ¬= 0;
       THEN:
....      RETURN WITH ERR_FLAG;
     END;

     /*    SPACE IN USER'S MEMORY MUST BE AVAILABLE                     */

     IF ((AST_CPL(ASTE#) & WIRED_DOWN_MASK) ¬= WIRED_DOWN) & (AST_SIZE(ASTE#) >
          PS_MEM_QUOTA);
       THEN:
....      RETURN WITH ERR_FLAG;
     END;
```

```
/*    SWAPIN IF NECESSARY                                              */

IF AST_ADR(ASTE#) = 0;
  THEN: SWAPIN(ASTE#);
END;

/*    REMOVE FROM SWAP CHAIN IF NECESSARY                              */

IF (AST_UNLOCK(ASTE#) & AST_UNLOCK_MASK) = AST_UNLOCK_ELAG;
  THEN: INDEX := 0;

     CYCLE
          NEXT := AST_SWAP_CHAIN(INDEX);
     .... EXIT WHEN NEXT = ASTE#;
          INDEX := NEXT;
     END;

     AST_SWAP_CHAIN(INDEX) := AST_SWAP_CHAIN(ASTE#);
     AST_DES_COUNT(ASTE#) := 0;
     AST_UNLOCK(ASTE#) := (AST_UNLOCK(ASTE#) & AST_LOCK_MASK);
END;

/*    DETERMINE TYPE OF ACCESS PERMITTED                               */

IE (AST_TYPE(ASTE#) & AST_TYPE_MASK) = AST_TYPE_DIRECTORY;
  THEN: MODE := 0; /* DIRECTORY ACCESSES MUST BE INTERPRETIVE */
  ELSE:
     IF (AST_WAL(ASTE#) & PS_PROCESS_MASK) = 0;
       THEN: MODE := SDR_READ_ACCESS;
       ELSE: MODE := SDR_WRITE_ACCESS;
     END;

END;

/*    LOAD SEGMENT DESCRIPTOR                                          */

INLINE(SPLHIGH);
LSD(ASTE#, PS_SDR_ADR + REG# + REG#, MODE);

/*    IF THIS IS THE CURRENT PROCESS LOAD HARDWARE REGS ALSO           */

IF PS_CURRENT_PROCESS = THE_CURRENT_PROCESS;
  THEN: SDR(P_REG#) := PS_SDR(REG#);
     SAR(P_REG#) := PS_SAR(REG#);
END;

INLINE(SPLLOW);

/*    INCREMENT DESCRIPTOR COUNT                                       */

AST_DES_COUNT(ASTE#) := AST_DES_COUNT(ASTE#) + 1;

/*    ADJUST USER'S QUOTA                                              */

IF (AST_CPL(ASTE#) & WIRED_DOWN_MASK) ¬= WIRED_DOWN;
  THEN: PS_MEM_QUOTA := PS_MEM_QUOTA - AST_SIZE(ASTE#);
END;

RC := OK_FLAG;
```

3.2.11 Disable (DISABLE)

The Disable CPC, DISABLE, is a user level external SKCPP function that is called by other user level external functions and by user level external programs with the parameter reg#. It is written in Project SUE System Language, including the Inline feature.

3.2.11.1 Description

DISABLE removes a segment from AS. It sets BLOCK# to PS_SAR(REG#). If BLOCK# equals zero, the register REG# contains no descriptor and DISABLE has no effect. If not, BLOCK# contains the storage address of the segment with the 6 least significant bits omitted. Since the MBT omits the 8 least significant bits of the address, two ASR (arithmetic shift right) commands are executed (using Inline code) on BLOCK#. DISABLE then finds the ASTE# of the segment. If BLOCK# is less than END BLOCK#, ASTE# is assigned MBT (BLOCK#). Otherwise, each value from ASTE#_MIN to ASTE#_MAX is tested until one is found such that AST_ADR(ASTE#) equals BLOCK#.

The priority level is then set high (using Inline code) and the descriptor destroyed by setting PS_SDR(REG#) and PS_SAR(REG#) to Ø. If PS_CURRENT_PROCESS is THE_CURRENT_PROCESS, the hardware segmentation registers are also cleared, as follows. If REG# is greater than CROSS_REG#, REG_CONSTANT is added to REG#. Next, if SDR(REG#) and SDR_CHANGE_MASK equals SDR_CHANGED, the change bit is set, and AST_CHANGE(ASTE#) is reset to AST_CHANGE(ASTE#) or AST_CHANGED. SDR(REG#) and SAR(REG#) are then zeroed and the priority level is set to low using Inline code.

AST_DES_COUNT(ASTE#) is then decremented, and if the segment is wired down, DISABLEing is complete. If, however, AST_CPL(ASTE#) and WIRED_DOWN_MASK equals Ø, the segment may be eligible for swapping out. If AST_DES_COUNT(ASTE#) is Ø, the segment is added to the head of the swap chain by setting AST_SWAP_CHAIN(ASTE#) to AST_SWAP_CHAIN(Ø) and AST_SWAP_CHAIN(Ø) to ASTE#. The segment is also unlocked in this case by resetting AST_UNLOCK(ASTE#) to the logical or of AST_UNLOCK (ASTE#) and AST_UNLOCK_FLAG,the DISABLE function is then completed by crediting PS_MEM_QUOTA with AST_SIZE(ASTE#).

Function: DISABLE

```
Parameters:  DISABLE(process#,reg#)
Effect:
IF PS_SAR(process#,reg#) ≠ 0;
THEN:  Let block# = 'PS_SAR'(process#,reg#);
   Let aste# = MBT_ASTE(block#);
   AST_CHANGE(block#) = 'AST_CHANGE'(block#)|
   'PS_SDR_CHANGE'(process#, reg#);
   PS_SAR(process#,reg#) = 0;
   PS_SDR(process#,reg#) = 0;
```

```
        AST_DES_COUNT(aste#) = 'AST_DES_COUNT'(aste#) - 1;
        IF (AST_DES_COUNT(aste#) = 0) &
           (AST_WIRED_DOWN(aste#) = OFF);
        THEN:  UNLOCK(aste#);
        END;
        IF AST_WIRED_DOWN(aste#) = OFF;
        THEN:  PS_MEM_QUOTA(process#) = 'PS_MEM_QUOTA'(process#) +
               size;
        END;
    END;
```

3.2.11.2 N/A

3.2.11.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
PCHECK	None
DCONNECT	

3.2.11.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function DISABLE. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
PS_SAR	REG#	BLOCK#
PS_SDR		ASTE#
PS_MEM_QUOTA		
PS_CURRENT_PROCESS		
AST_ADR		
AST_UNLOCK		
AST_SWAP_CHAIN		
AST_DES_COUNT		
AST_CPL		
AST_CHANGE		
MBT_ASTE		
THE_CURRENT_PROCESS		

Constants

ASR	REG_CONSTANT
AST_CHANGE	SDR_CHANGE_MASK
AST_UNLOCK_FLAG	SDR_CHANGED
ASTE#_MAX	SPLHIGH
ASTE#_MIN	SPLLOW
CROSS_REG#	WIRED_DOWN_MASK
END_BLOCK#	

3.2.11.5 Limitations

None.

3.2.11.6 Listing

```
DATA DISABLE(REG#);
     DECLARE
          WORD (BLOCK#, ASTE#);



PROGRAM DISABLE;

     /*    TRANSLATE REG# TO ASTE# (AND CHECK FOR DESCRIPTOR IN REGISTER)          */

     BLOCK# := PS_SAR(REG#);

     IF BLOCK# ¬= 0;
       THEN: INLINE(ASR, BLOCK#); /* THEIR BLOCKS TO MINE */
          INLINE(ASR, BLOCK#);

          IF BLOCK# < END_BLOCK#;
            THEN: ASTE# := MBT_ASTE(BLOCK#);
            ELSE:

               DO ASTE# := ASTE#_MIN TO ASTE#_MAX;
               .... EXIT WHEN AST_ADR(ASTE#) = BLOCK#;
               END;

          END;

          /*    DESTROY DESCRIPTOR                                                 */

          INLINE(SPLHIGH);
          PS_SDR(REG#) := 0;
          PS_SAR(REG#) := 0;

          /*    IF FOR CURRENT PROCESS ALSO CLEAR SEGMENTATION REGISTER            */

          IF PS_CURRENT_PROCESS = THE_CURRENT_PROCESS;
            THEN:
               IF REG# > CROSS_REG#;
                 THEN: REG# := REG# + REG_CONSTANT;
               END;

               /*    CHECK FOR CHANGE BIT BEING SET                                */
```

```
            IF (SDR(REG#) & SDR_CHANGE_MASK) = SDR_CHANGED;
              THEN: AST_CHANGE(ASTE#) := (AST_CHANGE(ASTE#) | AST_CHANGED);
            END;

            /*    CLEAR SEGMENTATION REGISTER                                    */

            SDR(REG#) := 0;
            SAR(REG#) := 0;
        END;

        INLINE(SPLLOW);

        /*    DECREMENT DESCRIPTOR COUNT                                         */

        AST_DES_COUNT(ASTE#) := AST_DES_COUNT(ASTE#) - 1;

        /*    THE FOLLOWING DOES NOT APPLY TO WIRED DOWN SEGMENTS                */

        IF (AST_CPL(ASTE#) & WIRED_DOWN_MASK) = 0;

          THEN:

            /*    IF NO DESCRIPTORS LEFT THEN UNLOCK AND ADD TO SWAP CHAIN       */

            IF (AST_DES_COUNT(ASTE#) = 0);
              THEN: AST_SWAP_CHAIN(ASTE#) := AST_SWAP_CHAIN(0);
                  AST_SWAP_CHAIN(0) := ASTE#;
                  AST_UNLOCK(ASTE#) := (AST_UNLOCK(ASTE#) | AST_UNLOCK_FLAG);
            END;

            /*    ADJUST MEMORY QUOTA                                            */

            PS_MEM_QUOTA := PS_MEM_QUOTA + AST_SIZE(ASTE#);
        END;

    END;
```

3.2.12 Outer P (OUTERP)

The Outer P CPC, OUTERP, is a user level external SKCPP function that is called by user level external programs with the parameter seg#. OUTERP calls only one kernel level internal function. It is written in Project SUE System Language, including the Inline feature.

3.2.12.1 Description

OUTERP performs security and implementation checks preliminary to calling P when P is invoked externally.

If AST_WAL(ASTE#) logical and PS_PROCESS_MASK equals zero, the process does not have write access to the segment, and OUTERP returns with ERR_FLAG.

It then performs INLINE(SPLHIGH) to set the priority level high. If SMFR_COUNT(ASTE#) equals -128, it is beyond the bounds of the implementation and ERR_FLAG is returned. Otherwise, P is called to

decrement the semaphore associated with the specified segment and block the process if the result is negative. RC is set to OK_FLAG.

```
Function: OUTERP
Parameters: OUTERP(aste#)
Effect:
IF AST_WAL(aste#,TCP);
THEN: P(aste#);
ELSE: RC(TCP) = NO;
END;
```

3.2.12.2 N/A

3.2.12.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
PCHECK	P

3.2.12.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function OUTERP. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
PS_PROCESS_MASK	ASTE#	RC
AST_WAL		
SMFR_COUNT		

Constants

ERR_FLAG
OK_FLAG
SPL_HIGH

3.2.12.5 Limitations

OUTERP returns ERR_FLAG if the process does not have write access to the segment or if the SMFR_COUNT for the segment equals

-128. Otherwise, RC=OK_FLAG. The SMFR_COUNT for an active segment runs from +127 to -128.

3.2.12.6 Listing

```
DATA OUTERP(ASTE#) RETURNS (RC);



PROGRAM OUTERP;

     /*     SECURITY CHECKS FIRST                                                         */

     /*     PROCESS MUST HAVE WRITE$READ ACCESS TO SEMAPHORE                              */

     IF (AST_WAL(ASTE#) & PS_PROCESS_MASK) = 0;
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     /*     IMPLEMENTATION CHECKS                                                         */

     INLINE(SPLHIGH);

     IF SMFR_COUNT(ASTE#) = - 128;
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     /*     CHECKING COMPLETE - PERFORM STATE CHANGE                                      */

     P(ASTE#);
     RC := OK_FLAG;
```

3.2.13 P (P)

The P CPC, P, is a kernel level internal SKCPP function that is called by both kernel level internal functions and one external user level function. P calls only kernel level internal functions. It is written in Project SUE System Language, including the Inline feature.

3.2.13.1 Description

P decrements a specified semaphore counter, and if the result is negative, blocks the process.

It blocks interrupts by setting the priority level high using Inline code, and then decrements SMFR_COUNT(SMFR#) by 1. If SMFR_COUNT(SMFR#) is less than zero, the process is added to the queue of processes blocked on this semaphore: PT_FLAGS(THE_CURRENT_PROCESS) is set to SMFR_POINTER(SMFR#) logical or BLOCKED and SMFR_POINTER(SMFR#) is set to THE_CURRENT_PROCESS.

If the current process has been blocked it also starts a new process running, as follows. If SMFR# is less than KERNEL_SMFR, the operation was performed on a segment semaphore. A V is performed on KERNEL_SMFR to prevent the possibility of a deadlock, SLEEP is called to find the next process ready to run, and a P is performed on KERNEL_SMFR to restore its previous condition. If SMFR# is greater than or equal to KERNEL_SMFR, P simply calls SLEEP. It then resets the priority level low using Inline code and returns control to the calling program.

```
Function:  P
Parameters:  P(smfr#)
Effect:
SMFR_COUNT(smfr#) = 'SMFR_COUNT'(smfr#) - 1;)
IF SMFR_COUNT(smfr#) < 0;
THEN:  PT_FLAGS(TCP) = BLOCKED;
   PT_LINK(TCP) = 'SMFR_POINTER'(smfr#);
   SMFR_POINTER(smfr#) = TCP;
END;
RC(TCP) = YES;
```

3.2.13.2 N/A

3.2.13.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
PCHECK	P
OUTERP	V
SWAPIN	SLEEP
SWAPOUT	
IPCRCV	
P	

3.2.13.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function P. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
PT_FLAGS SMFR_COUNT SMFR_POINTER THE_CURRENT_PROCESS	SMFR#	None

Constants

BLOCKED
KERNEL_SMFR
SEG#_FLAG
SPLHIGH
SPLLOW

3.2.13.5 Limitations

If the SMFR_COUNT for the specified SMFR# is less than Ø the process is blocked and added to the queue of processes blocked on that semaphore. The process becomes unblocked when enough V's are performed (see 3.2.15.1).

3.2.13.6 Listing

```
DATA P(SMFR#);


PROGRAM P;

     /*    BLOCK INTERRUPTS                                                    */

     INLINE(SPLHIGH);
     SMFR_COUNT(SMFR#) := SMFR_COUNT(SMFR#) - 1;

     /*    IF SEMAPHORE COUNT IS NEGATIVE, THEN PROCESS BECOMES BLOCKED        */

     IF SMFR_COUNT(SMFR#) < 0;
       THEN:

          /*    ADD CURRENT PROCESS TO QUEUE OF PROCESSES BLOCKED ON SEMAPHORE */

          PT_FLAGS(THE_CURRENT_PROCESS) := (SMFR_POINTER(SMFR#) | BLOCKED);
          SMFR_POINTER(SMFR#) := THE_CURRENT_PROCESS;

          /*    START A NEW PROCESS RUNNING                                    */

          IF SMFR# < KERNEL_SMFR;
            THEN: V(KERNEL_SMFR);
                SLEEP;
                P(KERNEL_SMFR);
            ELSE: SLEEP;
          END;

     END;

     INLINE(SPLLOW);
```

3.2.14 Outer V (OUTERV)

The Outer V CPC, OUTERV, is a user level external SKCPP function that is called by user level external programs with the parameter seg#. OUTERV calls one kernel level internal function. It is written in Project SUE System Language, including the Inline feature.

3.2.14.1 Description

OUTERV performs security and implementation checks whenever V is called externally. If AST_WAL(ASTE#) logical and PS_PROCESS_MASK equals Ø, the process lacks write access to the semaphore, so OUTERV returns with ERR_FLAG. To prevent interrupts, it sets the priority level high using Inline code. Next, if SMFR_COUNT(ASTE#) equals 127, it is beyond the bounds of the implementation, and OUTERV returns with ERR_FLAG. Otherwise it performs the state change, called V to increment the semaphore, and, if the result is non-positive, makes ready a process blocked on the semaphore. OUTERV returns with RC set to OK_FLAG.

```
Function:  OUTERV
Parameters:  OUTERV(aste#)
Effect:
IF AST_WAL(aste#,TCP);
THEN:  V(aste#);
ELSE:  RC(TCP) = NO;
END;
```

3.2.14.2 N/A

3.2.14.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
PCHECK	V

3.2.14.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function OUTERV. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
AST_WAL	ASTE#	RC
PS_PROCESS_MASK		
SMFR_COUNT		

Constants

ERR_FLAG
OK_FLAG
SPL_HIGH

3.2.14.5 Limitations

OUTERV returns ERR_FLAG if the process does not have write access to the semaphore or if the SMFR_COUNT for the segment equals 127.

3.2.14.6 Listing

```
DATA OUTERV(ASTE#) RETURNS (RC);



PROGRAM OUTERV;

     /*    SECURITY CHECKS FIRST                                          */

     /*    PROCESS MUST HAVE WRITE$READ ACCESS TO SEMAPHORE               */

     IF (AST_WAL(ASTE#) & PS_PROCESS_MASK) = 0;
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     /*    IMPLEMENTATION CHECKS                                          */

     INLINE(SPLHIGH);

     IF SMFR_COUNT(ASTE#) = 127;
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     /*    CHECKING COMPLETE - PERFORM STATE CHANGE                       */

     V(ASTE#);
     RC := OK_FLAG;
```

3.2.15 V (V)

The V CPC, V, is a kernel level internal SKCPP function that is called by both kernel level internal functions and one user level external function. It is written in Project SUE System Language, including the Inline feature.

3.2.15.1 Description

V is the inverse of P; it increments the specified semaphore counter and, if the result is non-positive, makes a blocked process ready. First, it prevents interrupts by setting the priority level high using Inline code. Then, it resets the SMFR_COUNT(SMFR#) to SMFR_COUNT(SMFR#) + 1.

If the SMFR_COUNT(SMFR#) is positive, the priority level is set low using Inline code and V is exited; if SMFR_COUNT(SMFR#) is less than or equal to zero, a blocked process must be unqueued. PROCESS_A is assigned the value of SMFR_POINTER(SMFR#). If SMFR_COUNT(SMFR#) does not equal Ø, V finds the process blocked longest by setting PROCESS_B equal to PROCESS_A and PROCESS_A equal to PT_LINK(PROCESS_B) until PT_LINK(PROCESS_A) equals zero. It then removes PROCESS_A from the end of the queue by setting PT_LINK(PROCESS_B) equal to Ø. Otherwise, if SMFR_COUNT(SMFR#) equals Ø, only PROCESS_A is blocked on the semaphore; V removes it by setting SMFR_POINTER(SMFR#) to Ø.

V then readies PROCESS_A by assigning PT_FLAGS(PROCESS_A) the value READY. It then sets the priority level low using Inline code and returns.

```
Function:  V
Parameters:  V(smfr#)
Effect:  SMFR_COUNT(smfr#) = 'SMFR_COUNT'(smfr#) + 1;
IF SMFR_COUNT(smfr#)  <= 0;
THEN:
   IF SMFR_COUNT(smfr#) = 0;
   THEN:  Let process# = 'SMFR_POINTER'(smfr#);
      SMFR_POINTER(smfr#) = 0;
   ELSE:  Let process# = VEND;
      VUNCHAIN('SMFR_POINTER'(smfr#));
   END:
   PT_FLAGS(process#) = READY;
END;
RC(TCP) = YES;
```

```
Function:  VEND
Parameters:  VEND(process#)
Value:
IF 'PT_LINK'(process#) = 0;
THEN:  process#;
ELSE:  VEND('PT_LINK'(process#));
END;

Function:  VUNCHAIN
Parameters:  VUNCHAIN(process#)
Effect:
IF 'PT_LINK'('PT_LINK'(process#)) = 0;
THEN:  PT_LINK(process#) = 0;
ELSE:  VUNCHAIN('PT_LINK'(process#));
END;
```

3.2.15.2 N/A

3.2.15.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
PCHECK	None
OUTERV	
IPCRCV	
STOPP	
P	

3.2.15.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function V. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
PT_LINK	SMFR#	PROCESS_A
PT_FLAG		PROCESS_B
SMFR_COUNT		
SMFR_POINTER		

Constants

READY
SPLHIGH
SPLLOW

3.2.15.5 Limitations

None.

3.2.15.6 Listing

```
DATA V(SMFR#);
     DECLARE
          WORD (PROCESS_A, PROCESS_B);



PROGRAM V;

     /*    BLOCK INTERRUPTS                                                    */

     INLINE(SPLHIGH);
     SMFR_COUNT(SMFR#) := SMFR_COUNT(SMFR#) + 1;

     /*    IF SEMAPHORE COUNT IS NON-POSITIVE, THEN A PROCESS BLOCKED ON THE SEMAPHORE  *
      * MUST BE UNQUEUED                                                      */

     IF SMFR_COUNT(SMFR#)<= 0;
       THEN:

          /*    THE MOST RECENT PROCESS ADDED TO THIS SEMAPHORE'S QUEUE        */

          PROCESS_A := SMFR_POINTER(SMFR#);

          /*    IF THERE IS MORE THAN ONE PROCESS ON QUEUE, FOLLOW CHAIN THROUGH  *
           * PROCESS TABLE                                                   */

          IF SMFR_COUNT(SMFR#) ¬= 0;
            THEN:

               CYCLE
               .... EXIT WHEN PT_LINK(PROCESS_A) = 0;
                    PROCESS_B := PROCESS_A;
                    PROCESS_A := PT_LINK(PROCESS_B);
               END;

               /*    REMOVE PROCESS_A FROM END OF QUEUE                     */

               PT_LINK(PROCESS_B) := 0;
            ELSE: SMFR_POINTER(SMFR#) := 0;
          END;

          /*    PROCESS_A BECOMES READY                                      */

          PT_FLAGS(PROCESS_A) := READY;
     END;

     INLINE(SPLLOW);
```

3.2.16 Send Interprocess Communication (IPCSEND)

The Send Interprocess Communication CPC, IPCSEND, is a user level external SKCPP function that is called by user level external programs and one user level external function with the parameters process# and message. It is written in Project SUE System Language.

3.2.16.1 Description

IPCSEND sends a message to a specified process if security and implementation constraints are met. If PT_FLAGS(PROCESS#) logical and PT_FLAGS_MASK equals INACTIVE, IPCSEND returns. It has no return code since that might be used to compromise security.

If PS_CURRENT_PROCESS does not equal EXEC_PROCESS#, it is not trusted and the following security checks must be made to preserve the *-property. If PT_CURRENT_CLASS(PROCESS#) is less than PS_CUR_CLASS or if PT_CUR_CAT(PROCESS#) does not equal PT_CUR_CAT (PROCESS#) logical or PS_CUR_CAT, IPCSEND returns with no effect.

If PT_IPC_QUOTA(PROCESS#) is zero, IPCSEND ignores the call because the receiving process has no more free IPC elements.

Checking complete, it then allocates an element from the queue of free elements by assigining INDEX the value of IPC_LINK(Ø), and resetting IPC_LINK(Ø) to IPC_LINK(INDEX) and IPC_LINK(INDEX) to Ø. It fills in the element by letting IPC_PROCESS#(INDEX) equal PS_CURRENT_PROCESS logical or DOMAIN and letting IPC_DATA(INDEX) equal MESSAGE.

If PT_IPC_QUEUE_HEAD(PROCESS#) logical and BYTE_MASK equals IPC_WAIT, the specified process is blocked because it is awaiting a message and none was available. In this case, PT_IPC_QUEUE_HEAD (PROCESS#) is set to INDEX and the process is unblocked by setting PT_FLAGS(PROCESS#) to READY. Otherwise, the new IPC element must be appended to the process's queue of messages. If PT_IPC_QUEUE_HEAD (PROCESS#) equals zero, the process's IPC queue is empty, and PT_IPC_QUEUE_HEAD(PROCESS#) is set to INDEX. If not, to find the end of the queue, INDEX2 is set equal to PT_IPC_QUEUE_HEAD(PROCESS#). Until IPC_LINK(INDEX2) equals zero, INDEX2 is reset to IPC_LINK (INDEX2). Since INDEX2 now holds the IPC element number of the last element in the queue, letting IPC_LINK(INDEX2) equal INDEX attaches the new element to the end of the queue.

Finally, the IPC element quota of the receiving process is adjusted by decrementing PT_IPC_QUOTA(PROCESS#) and IPCSEND is exited.

```
Function:  IPCSEND
Parameters:  IPCSEND(process#,message,domain)
Effect:
IF (PT_FLAGS(process#) ≠ INACTIVE) &
   (((PS_CUR_CLASS(process#) ≯ = PS_CUR_CLASS(TCP)) &
   (PS_CUR_CAT(process#) ⊉ PS_CUR_CAT(TCP)))|
   (PT_TYPE(TCP) = TRUSTED)) &
   ('PT_IPC_QUOTA'(process#)  ≠ 0);
THEN:  Let ipce# = 'IPC_LINK'(0);
   IPC_LINK(0) = 'IPC_LINK'(ipce#);
   IPC_LINK(ipce#) = 0;
   IPC_PROCESS(ipce#) = TCP;
   IPC_DOMAIN(ipce#) = domain;
   IPC_DATA(ipce#) = message;
   IF 'PT_IPC_QUEUE_HEAD'(process#) = 0;
   THEN:  PT_IPC_QUEUE_HEAD(process#) = ipce#;
   ELSE:  Let eipce# = FINDIPCEND('PT_IPC_QUEUE_HEAD'(process#));
      IPC_LINK(eipce#) = ipce#;
   END;
   PT_IPC_QUOTA(process#) = 'PT_IPC_QUOTA'(process#) - 1;
   IF  'PT_IPC_WAIT'(process#) = NO;
   THEN:  PT_IPC_WAIT(process#) = OFF;
   PT_FLAGS(process#) = READY;
   END;
END;

Function:  FINDIPCEND
Parameters:  FINDIPCEND(ipce#)
Value:
IF IPC_LINK(ipce#) = 0;
THEN:  ipce#;
ELSE:  FINDIPCEND(IPC_LINK(ipce#));
END;
```

3.2.16.2 N/A

3.2.16.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
PCHECK STOPP	None

3.2.16.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function IPCSEND. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
PT_FLAGS	PROCESS#	INDEX
PT_CUR_CLASS	MESSAGE	INDEX2
PT_CUR_CAT	DOMAIN	
PT_IPC_QUOTA		
PT_IPC_QUEUE_HEAD		
PS_CURRENT_PROCESS		
IPC_LINK		
IPC_PROCESS#		
IPC_DATA		

Constants

EXEC_PROCESS#
INACTIVE
IPC_WAIT
PT_FLAGS_MASK
READY

3.2.16.5 Limitations

If the PT_IPC_QUOTA for the process is zero, IPCSEND ignores the call. IPCSEND returns with no return code if PS_CURRENT_PROCESS is not equal to EXEC_PROCESS#, if PT_CURRENT_CLASS(PROCESS#) is less than PS_CUR_CLASS, if PT_CUR_CAT(PROCESS#) does not equal PT_CUR_CAT (PROCESS#) logical or PS_CUR_CAT, or if PT_FLAGS and PT_FLAGS_MASK equals INACTIVE.

3.2.16.6 Listing

```
DATA IPCSEND(PROCESS#, MESSAGE, DOMAIN);
```

```
PROGRAM IPCSEND;
     DECLARE
          WORD (INDEX, INDEX2);

     IF (PT_FLAGS(PROCESS#) & PT_FLAGS_MASK) = INACTIVE;
       THEN:
....      RETURN;
     END;

     /*     SECURITY CHECK                                                  */

     IF PS_CURRENT_PROCESS ¬= EXEC_PROCESS#;
       THEN:
          IF PT_CUR_CLASS(PROCESS#) < PS_CUR_CLASS;
            THEN:
....           RETURN;
          END;

          IF PT_CUR_CAT(PROCESS#) ¬= (PT_CUR_CAT(PROCESS#) | PS_CUR_CAT);
            THEN:
....           RETURN;
          END;

     END;

     /*     IMPLEMENTATION CHECK                                            */

     IF PT_IPC_QUOTA(PROCESS#) = 0;
       THEN:
....      RETURN;
     END;

     /*     ALLOCATE AN IPC QUEUE ELEMENT                                   */

     INDEX := IPC_LINK(0);
     IPC_LINK(0) := IPC_LINK(INDEX);
     IPC_LINK(INDEX) := 0;

     /*     FILL IN IPC ELEMENT                                             */

     IPC_PROCESS#(INDEX) := (PS_CURRENT_PROCESS | DOMAIN);
     IPC_DATA(INDEX) := MESSAGE;

     /*     IS PROCESS WAITING?                                             */

     IF (PT_IPC_QUEUE_HEAD(PROCESS#) & BYTE_MASK) = IPC_WAIT;
       THEN: PT_IPC_QUEUE_HEAD(PROCESS#) := INDEX;
          PT_FLAGS(PROCESS#) := READY;
       ELSE:
          IF PT_IPC_QUEUE_HEAD(PROCESS#) = 0;
            THEN: PT_IPC_QUEUE_HEAD(PROCESS#) := INDEX;
            ELSE: INDEX2 := PT_IPC_QUEUE_HEAD(PROCESS#);

               CYCLE
               .... EXIT WHEN IPC_LINK(INDEX2) = 0;

                    INDEX2 := IPC_LINK(INDEX2);
               END;

               IPC_LINK(INDEX2) := INDEX;
          END;

     END;

     /*     ADJUST QUOTA                                                    */

     PT_IPC_QUOTA(PROCESS#) := PT_IPC_QUOTA(PROCESS#) - 1;
```

3.2.17 Receive Interprocess Communication (IPCRCV)

The Receive Interprocess Communication CPC, IPCRCV, is a user level external SKCPP function that is called by user level external programs. IPCRCV calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.17.1 Description

IPCRCV receives an interprocess communication message. If PT_IPC_QUEUE_HEAD(PR_CURRENT_PROCESS) equals Ø, there are no messages on queue. In this case, IPCRCV sets PT_IPC_QUEUE_HEAD(PS_CURRENT_PROCESS) to IPC_WAIT and PT_FLAGS(PS_CURRENT_PROCESS) to BLOCKED. This prevents future allocation of the processor to the current process. Then, to prevent a deadlock, it calls V to increment the kernel semaphore and then calls SLEEP to find and execute the next process that is ready. When a message is available and the process is unblocked and running, it performs a P on the kernel semaphore to restore its original value.

Now, to remove the IPC element from the head of queue, INDEX is set to PT_IPC_QUEUE_HEAD(PS_CURRENT_PROCESS) and PT_IPC_QUEUE_HEAD (PS_CURRENT_PROCESS) is set to IPC_LINK(INDEX).

IPCRCV can now use the information from the IPC message element. It assigns RC the sending process# and domain held in IPC_PROCESS# (INDEX) and KRC2 gets the message held in IPC_DATA(INDEX).

The IPC element is then put back on the free chain and the process' quota is credited. IPC_LINK(INDEX) is set equal to IPC_LINK(Ø) and IPC_LINK(Ø) is set to INDEX. IPCRCV then increments PT_IPC_QUOTA(PS_CURRENT_PROCESS) to conclude the operation.

```
Function:   IPCRCV
Parameters:   IPCRCV
Effect:
IF 'PT_IPC_QUEUE_HEAD'(TCP) = 0;
THEN:  PT_IPC_WAIT(TCP) = ON;
   PT_FLAGS(TCP) = BLOCKED;
   IPCRCV2;
ELSE:  IPCUNQUEUE;
END;
```

```
Function:  IPCUNQUEUE
Parameters:  IPCUNQUEUE
Effect:
Let ipce# = 'PT_IPC_QUEUE_HEAD'(TCP);
PT_IPC_QUEUE_HEAD(TCP) = 'IPC_LINK'(ipce#);
RC(TCP) = IPC_PROCESS(ipce#), IPC_DOMAIN(ipce#),IPC_DATA(ipce#);
IPC_LINK(ipce#) = 'IPC_LINK'(0);
IPC_LINK(0) = ipce#;
PT_IPC_QUOTA(TCP) = 'PT_IPC_QUOTA'(TCP) + 1;

Function:  IPCRCV2
Parameters:  IPCRCV2
Effect:
IF 'PT_IPC_QUEUE_HEAD'(TCP) ≠ 0;
THEN:  IPCUNQUEUE;
END;
```

3.2.17.2 N/A

3.2.17.3 <u>Interfaces</u>

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
PCHECK	P
	V
	SLEEP

3.2.17.4 <u>Data Organization</u>

Listed below are Security Kernel data base references and constants used by the function IPCRCV. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

<u>Data Base References</u>

Global References	Function Parameters	Local References
PT_IPC_QUEUE_HEAD	None	INDEX
PT_FLAGS		RC
PT_IPC_QUOTA		
PS_CURRENT_PROCESS		
IPC_LINK		
IPC_PROCESS#		
IPC_DATA		

Constants

BLOCKED
IPC_WAIT
KERNEL_SMFR

3.2.17.5 Limitations

None

3.2.17.6 Listing

```
DATA IPCRCV RETURNS (RC);
     DECLARE
          WORD (INDEX);



PROGRAM IPCRCV;

     /*    NO SECURITY CHECKING                                                      */

     /*    ANYTHING THERE                                                            */

     IF PT_IPC_QUEUE_HEAD(PS_CURRENT_PROCESS) = 0;
       THEN: PT_IPC_QUEUE_HEAD(PS_CURRENT_PROCESS) := IPC_WAIT;
          PT_FLAGS(PS_CURRENT_PROCESS) := BLOCKED;
          V(KERNEL_SMFR);
          SLEEP;
          P(KERNEL_SMFR);
     END;

     /*    REMOVE FIRST MESSAGE ELEMENT                                              */

     INDEX := PT_IPC_QUEUE_HEAD(PS_CURRENT_PROCESS);
     PT_IPC_QUEUE_HEAD(PS_CURRENT_PROCESS) := IPC_LINK(INDEX);

     /*    TAKE STUFF OUT OF IPC MESSAGE ELEMENT                                     */

     RC := IPC_PROCESS#(INDEX);
     KRC2 := IPC_DATA(INDEX);

     /*    PUT BACK ON FREE CHAIN AND INCREMENT QUOTA                                */

     IPC_LINK(INDEX) := IPC_LINK(0);
     IPC_LINK(0) := INDEX;
     PT_IPC_QUOTA(PS_CURRENT_PROCESS) := PT_IPC_QUOTA(PS_CURRENT_PROCESS) + 1;
```

3.2.18 Stop Process (STOPP)

The Stop Process CPC, STOPP, is a user level external SKCPP function that is called by user level external programs. STOPP calls both user level external functions and kernel level internal functions. It is written in Project SUE System Language.

3.2.18.1 Description

STOPP terminates a user's ownership of a process. It loops with I going from SEG#_MIN to SEG#_MAX, setting ASTE# equal to PS_SEG(I). Then, if ASTE# logical and SEG_FLAG equals Ø, ASTE# indeed contains an AST entry number, so segment I is in the process's WS. If so, STOPP calls DCONNECT to release any segments to which the process has access.

It then sets I equal to PT_IPC_QUEUE_HEAD(PS_CURRENT_PROCESS). If I does not equal Ø, the IPC queue must be cleared. To find the last element in the IPC queue, until IPC_LINK(I) equals Ø, I is reset to IPC_LINK(I). The entire IPC queue is inserted at the head of the free element chain by letting IPC_LINK(I) equal IPC_LINK(Ø), IPC_LINK(Ø) equal PT_IPC_QUEUE_HEAD(PS_CURRENT_PROCESS), and PT_IPC_QUEUE_HEAD(PS_CURRENT_PROCESS) and PT_IPC_QUEUE_HEAD (PS_CURRENT_PROCESS) equal Ø.

Next, STOPP prepares to remove the process's kernel stack. It sets ASTE# to PT_KS_ASTE#(PS_CURRENT_PROCESS) which holds the aste# of the process's kernel stack. It adds this to the chain of segments eligible to be swapped out by letting AST_SWAP_CHAIN(ASTE#) equal AST_SWAP_CHAIN(Ø), and letting AST_SWAP_CHAIN(Ø) equal ASTE#. It also resets AST_UNLOCK(ASTE#) to AST_UNLOCK(ASTE#) logical or AST_UNLOCK_FLAG.

In order to inform the executive process of the current process's termination, STOPP calls IPCSEND. It then sets PT_FLAGS(PS_CURRENT_PROCESS) to INACTIVE, performs a V on the kernel semaphore to prevent deadlocking, and calls SLEEP to allocate the processor to a ready process.

Function: STOPP
Parameters: STOPP(process#)
Effect:
(∀seg#)
IF (SEG#_MIN ≤ seg# ≤ SEG_MAX) &
 PS_SEG_INUSE(process#,seg#);
THEN: DCONNECT(process#,PS_SEG(process#,seg#),seg#);
END;

```
IF  'PT_IPC_QUEUE_HEAD'(process#) ≠ 0
THEN:  Let ipce# = FINDIPSCEND('PT_IPC_QUEUE_HEAD(process#));
IPC_LINK(ipce#) = 'IPC_LINK' (0);
IPC_LINK(0) = 'PT_IPC_QUEUE_HEAD'(process#);
PT_IPC_QUEUE_HEAD(process#) = 0;
END:
PT_FLAG (process#) = INACTIVE;
IPCSEND(EXECUTIVE_PROCESS#,0,KERNEL_DOMAIN);
SLEEP;
```

3.2.18.2 N/A

3.2.18.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
PCHECK	DCONNECT
	IPCSEND
	V
	SLEEP

3.2.18.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function STOPP. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1 For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
PT_FLAGS	None	ASTE#
PT_KS_ASTE#		I
PT_IPC_QUEUE_HEAD		
PS_CURRENT_PROCESS		
AST_SWAP_CHAIN		
AST_UNLOCK		
IPC_LINK		

Constants

AST_UNLOCK_FLAG
EXEC_PROCESS#
INACTIVE
KERNEL_DOMAIN

KERNEL_SMFR
SDR_READ_ACCESS
SEG_FLAG
SEG#_MAX
SEG#_MIN

3.2.18.5 Limitations

None

3.2.18.6 Listing

```
DATA STOPP;



PROGRAM STOPP;
     DECLARE
          WORD (I, ASTE#, DUMMY);

     /*    CLEAR OUT "B"                                                  */

     DO I := SEG#_MIN TO SEG#_MAX;
          ASTE# := PS_SEG(I);

          IF (ASTE# & SEG_FLAG) = 0;
            THEN: DCONNECT(I, ASTE#);
          END;

     END;

     /*    CLEAR OUT IPC QUEUE                                            */

     I := PT_IPC_QUEUE_HEAD(PS_CURRENT_PROCESS);

     IF I ¬= 0;
       THEN:

          CYCLE
          .... EXIT WHEN IPC_LINK(I) = 0;
               I := IPC_LINK(I);
          END;

          IPC_LINK(I) := IPC_LINK(0);
          IPC_LINK(0) := PT_IPC_QUEUE_HEAD(PS_CURRENT_PROCESS);
          PT_IPC_QUEUE_HEAD(PS_CURRENT_PROCESS) := 0;
     END;

     /*    GET RID OF K STACK                                             */

     ASTE# := PT_KS_ASTE#(PS_CURRENT_PROCESS);
     AST_SWAP_CHAIN(ASTE#) := AST_SWAP_CHAIN(0);
     AST_SWAP_CHAIN(0) := ASTE#;
     AST_UNLOCK(ASTE#) := (AST_UNLOCK(ASTE#) | AST_UNLOCK_FLAG);

     /*    LET EXEC KNOW WHAT'S HAPPENING                                 */

     IPCSEND(EXEC_PROCESS#, 0, KERNEL_DOMAIN);
     PT_FLAGS(PS_CURRENT_PROCESS) := INACTIVE;
     V(KERNEL_SMFR);
     SLEEP;
```

3.2.19 Read Directory (READIR)

The Read Directory CPC, READIR, is a user level external SKCPP function that is called by user level external programs with the parameters aste# and offset. READIR calls only kernel level internal functions. It is written in the Project SUE System Language.

3.2.19.1 Description

READIR gives interpretive read access to an entry in a directory in the process's address space.

If AST_TYPE(ASTE#) logical and AST_TYPE_MASK does not equal AST_TYPE_DIRECTORY the specified segment is not a directory, so READIR returns with ERR_FLAG.

Next, READIR must gain access to the directory. If AST_ADR (ASTE#) is zero, the directory is not present in main memory. SWAPIN is called to correct this situation. LSD is then invoked to load the segment descriptors. If DIR_SIZE(OFFSET) equals zero, READIR returns with ERR_FLAG; otherwise checking is complete and the data in the directory that is at the security level of the directory can be returned. CLASS, CAT, SEG_TYPE, and SIZE are assigned the value of DIR_CLASS(OFFSET) logical and DIR_CLASS_MASK, DIR_CAT(OFFSET), DIR_TYPE(OFFSET) logical and DIR_TYPE_MASK, and DIR_SIZE(OFFSET), respectively. READIR then regains access to the user's SRØ stack by assigning to KSDR3 and KSAR3 the values of SDRØ and SARØ. It then inserts the data, letting CLASS_APARM, CAT_APARM, SEG_TYPE_APARM, and SIZE_APARM equal CLASS, CAT, SET_TYPE, and SIZE. READIR then returns with RC set to OK_FLAG.

```
Function:  READIR
Parameters:  READIR(process#,aste#,offset)
Effect:
IF (AST_TYPE(aste#) = DIRECTORY &
   (DIR_SIZE(aste#,offset) ≠ 0
THEN:  RC(process#) = DIR_TYPE(aste#,offset),
                      DIR_CLASS(aste#, offset),
                      DIR_CAT(aste#, offset),
                      DIR_SIZE(aste#, offset);
ELSE:  RC(process#) = NO;
END;
```

3.2.19.2 N/A

3.2.19.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
PCHECK	SWAPIN LSD

3.2.19.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function READIR. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
AST_TYPE	ASTE#	CLASS
AST_ADR	OFFSET	CAT
DIR_CLASS		SEG_TYPE
DIR_CAT		SIZE
DIR_TYPE		RC
DIR_SIZE		
CLASS_ALARM		
CAT_APARM		
SEG_TYPE_APARM		
SIZE_APARM		
SDR & SAR		

Constants

AST_TYPE_DIRECTORY
AST_TYPE_MASK
DIR_CLASS_MASK
DIR_KSR_ADR
DIR_TYPE
DIR_TYPE_MASK
ERR_FLAG
OK_FLAG
SDR_READ_ACCESS

3.2.19.5 Limitations

READIR returns ERR_FLAG if the specified segment is not a directory or if the specified offset is not in main memory. Otherwise, RC = OK_FLAG.

3.2.19.6 Listing

```
DATA READIR(ASTE#, OFFSET) RETURNS (RC);



PROGRAM READIR;
     DECLARE
          WORD (CLASS, CAT, SEG_TYPE, SIZE);

     /*    IMPLEMENTATION CHECKS                                             */

     /*    CHECK THAT SPECIFIED SEGMENT IS A DIRECTORY                       */

     IF (AST_TYPE(ASTE#) & AST_TYPE_MASK) ¬= AST_TYPE_DIRECTORY;
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     /*    GAIN ACCESS TO THE DIRECTORY                                      */

     IF AST_ADR(ASTE#) = 0;
       THEN: SWAPIN(ASTE#);
     END;

     LSD(ASTE#, DIR_KSR_ADR, SDR_READ_ACCESS);

     /*    CHECK THAT SPECIFIED OFFSET EXISTS                                */

     IF DIR_SIZE(OFFSET) = 0;
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     /*    SAVE CLASS, CAT, SEG_TYPE, AND SIZE                               */

     CLASS := DIR_CLASS(OFFSET) & DIR_CLASS_MASK;
     CAT := DIR_CAT(OFFSET);
     SEG_TYPE := DIR_TYPE(OFFSET) & DIR_TYPE_MASK;
     SIZE := DIR_SIZE(OFFSET);

     /*    REGAIN ACCESS TO USERS SRO STACK AND INSERT DATA                  */

     KSDR3 := SDR0;
     KSAR3 := SAR0;
     CLASS_APARM := CLASS;
     CAT_APARM := CAT;
     SEG_TYPE_APARM := SEG_TYPE;
     SIZE_APARM := SIZE;
     RC := OK_FLAG;
```

3.2.20 Start Process (STARTP)

The Start Process CPC, STARTP, is a user level external SKCPP function that is called by only one user level external program, the Executive Process with the parameters process# user, project, class, cat, proc_offset#, and new_process#. STARTP calls both user level external functions and kernel level internal functions. It is written in Project SUE System Language, including the Inline feature.

3.2.20.1 Description

STARTP initializes a new process when invoked by the Executive Process. If PS_CURRENT_PROCESS does not equal EXEC_PROCESS#, it returns with ERR_FLAG. Also, if PT_FLAGS(PROCESS#) logical and PT_FLAGS_MASK (the FLAGS and LINK entries share a byte) is not INACTIVE, it returns ERR_FLAG.

STARTP then calls LSD to load the segment descriptors of the new process's process segment so the PS can be initialized. It sets PS_CURRENT_PROCESS to PROCESS#, PS_USER_ID to USER, PS_PROJECT_ID to PROJECT, PS_CUR_CLASS and also PT_CUR_CLASS(PROJECT#) to CLASS, PS_CUR_CAT and also PT_CUR_CAT(PROCESS#) to CAT, PS_MEM_QUOTA to MEM_QUOTA, and PT_IPC_QUOTA to IPC_QUOTA. STARTP uses Inline code to find MASK and NOTMASK, it moves the process# to register 3, decrements it, and negates it. Then, registers Ø and 1 are set to $4ØØØ_{16}$ (all Ø's except the second bit, bit 14) and $BFFF_{16}$ (all 1's except the second bit), respectively. It then performs an ASH to shift arithmetically the contents of registers Ø and 1 N places, where N is the number in register 3. If N is positive, a left shift is performed, and the low order bits are filled in with Ø's; if N is negative, a right shift is performed and bit 15 is replicated. Since register 3 contains 1 - PROCESS#, the result in register Ø is a 1 in the bit corresponding to the process number, and Ø's elsewhere, the leftmost bit representing process Ø and the rightmost, process 15. Register 1 contains the one's complement of register Ø except that for process Ø it contains Ø's in both the leftmost and the rightmost bits. Register Ø is moved to PS_PROCESS_MASK and register 1 is moved to PS_PROCESS_NOTMASK, which are used in accessing AST_CPL and AST_WAL. STARTP then sets each PS_SAR(I) and PS_SDR(I) to Ø, as I goes from Ø to 15. Also, with I starting at Ø until I equals SEG#_MAX, PS_SEG(I) is set equal to I + 1 logical or SEG_FLAG, placing all segment numbers on the free segment chain. Assigning PS_SEG(SEG#_MAX) the value SEG_FLAG marks the end of the free segment chain and completes the insertion of the process segment information.

STARTP then puts ROOT in the access space of the new process. It takes segment 1 from the free chain by letting PS_SEG(Ø) = PS_SEG(1) and assigned to it ROOT_ASTE#. It then connects the new process to ROOT by resetting AST_CPL(ROOT_ASTE#) to AST_CPL(ROOT_ASTE#) logical or PS_PROCESS_MASK.

Now, access to user and kernel stacks are provided for the new process. GETR is called to gain read access to the process directory directory segment specified by ROOT_ASTE#, PDD_OFFSET; the seg# returned is assigned to PDD_SEG#. Next, read access is gained to the executive's process directory using GETR; similarly, the segment number returned is assigned to PD_SEG#. GETW is now called to provide write access to a user stack identified by an offset into the process directory of PROCESS#. STARTP assigns the seg# GETW returns to SS_SEG#. This segment is then ENABLEd. To get write access to the kernel stack is more difficult because GETW would fail. First, the segment descriptors of the process directory are loaded with LSD. STARTP then sets KS_ASTE# to the aste# of the PROCESS#_MAX + PROCESS# entry to the process directory: DIR_DISK holds the disk address of the entry and HASH converts this to an AST entry number. If KS_ASTE# is Ø, STARTP halts. Otherwise, the segment descriptors of the segment identified by KS_ASTE# are loaded, the AST_DES_COUNT(KS_ASTE#) is incremented, and PT_KS_ASTE#(PROCESS#) is set equal to KS_ASTE#.

STARTP now calls DCONNECT to release from the WS some intermediate directories, the process directory directory and the process directory. It invokes GETR to gain read access to the code directory identified by ROOT_ASTE#, CD_OFFSET and assigns its segment number to CD_SEG#. Next, it calls GETR to gain access to the segment, PS_SEG(CD_SEG#), PROC_OFFSET, and assigns its segment number to PROC_SEG#. A call to ENABLE places PROC_SEG#, which contains the new process's initial code segment, in the AS. The code directory, CD_SEG# can now be released by DCONNECT.

STARTP then calls LSD to switch back to the executive process by loading the descriptors of its process segment in the register at PS_KSR_ADR. LSD is called again to load the segment descriptors of the new process to give the executive process write access to its process segment.

Setting PT_R5(PROCESS#), a general register, equal to zero, PT_FLAGS(PROCESS#) to READY, and PT_IPC_QUEUE_HEAD to zero, completes the initialization of the new segment. STARTP returns with RC equal to OK_FLAG.

```
Function:  STARTP
Parameters:  STARTP(process#,user,project,class,cat,new_process#,
             proc_offset#);
Effect:
IF  (process# ≠ EXECUTIVE_PROCESS#) 1
    (PT_FLAGS(new_process#) ≠ INACTIVE);
THEN:  RC(process#) = NO;
ELSE:  PS-USER_ID(new_process#) = user;
   PS_PROJECT_ID(new_process#) = project;
   PS_CLASS(new_process#) = class;
   PS_CAT(new-process#) = cat;
   PS_MEM_QUOTA(new_process#) = MEM_QUOTA;
   PS_IPC(new_process#) = IPC_QUOTA;
   (∀reg#)
   IF  (REG#_MIX ≤ reg# ≤ REG#_MAX);
   THEN:  PS_SAR(new_process#,reg#) = 0;
      PS_SDR(new_process#,reg#) = 0;
   END;
   (∀seg#)
      IF  (SEG#_MIX ≤ seg# ≤ SEG#_MAX);
      THEN:  PS_SEG_INUSE(new_process#,seg#) = FALSE;
         PS_SEG(new_process#,seg#) =
         (seg#+1)MODULO(SEG#_MAX+1);
      END;
      PS_SEG(new_process#,0) = PS_SEG(new_process#,1);
      PS_SEG(new_process#,1) = ROOT_ASTE#;
      AST_CPL(ROOT_ASTE#,new_process#) = TRUE:
      GETR(new_process#,ROOT_ASTE,PDD_OFFSET#);
      Let pdd_seg# = RC(new_process#);
      GETR(new_process#,PS_SEG(new_process#,pdd_seg#),
          EXECUTIVE_PROCESS#);
      Let pd_seg# = RC(new_process#);
      GETW(new_process#, PS_SEG(new_process#,pd_seg#),
          new_process#);
      Let stack_seg# = RC(new_process#);
          ENABLE(new_process#, PS_SEG(new_process#,
                stack_seg#), STACK_REG#);
          DCONNECT(new_process#,PS_SEG(new_process#,
                  pdd_seg#),pdd_seg#);
          GETR(new_process#,ROOT_ASTE#,CD_OFFSET#);
      Let cd_seg# = RC(new_process#);
          GETR(new_process#,PS_SEG(new_process#,
              cd_seg#),proc_offset#);
      Let proc_seg# = RC(new_process#);
          ENABLE(new_process#,PS_SEG(new_process#,
                proc_seg#)PROC_REG#);
```

```
          DCONNECT(new_process#,PS_SEG(new_process#,
                  cd_seg#),cd_seg#);
          PT_IPC_QUEUE_HEAD(new_process#) = 0;
          PT_FLAGS(new_process#) = READY;
     END;
```

3.2.20.2 N/A

3.2.20.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
PCHECK	GETR
	GETW
	DCONNECT
	ENABLE
	HASH
	LSD

3.2.20.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function STARTP. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Local Parameters	Local References
PS_CURRENT_PROCESS	USER	PDD_SEG#
PS_USER_ID	PROJECT	PD_SEG#
PS_PROJECT_ID	CLASS	CD_SEG#
PS_CUR_CLASS	CAT	SS_SEG#
PS_CUR_CAT	PROCESS#	KS_ASTE#
PS_MEM_QUOTA	PROC_OFFSET	PROC_SEG#
PS_PROCESS_MASK		I
PS_PROCESS_NOTMASK		DUMMY
PS_SAR		RC
PS_SDR		
PS_SEG		
PS_FLAG		
PT_FLAG		
PT_IPC_QUOTA		

Global References	Local Parameters	Local References
PT_KSDR2		
PT_R5		
PT_PS_ASTE#		
PT_KSDR1		
PT_IPC_QUEUE_HEAD		

Constants

ASHROR3	OK_FLAG
ASHR1R3	PDD_OFFSET
CD_OFFSET	PROCESS#_MAX
DEC	PS_KSR_ADR
DIR_KSR_ADR	PT_FLAGS_MASK
ERR_FLAG	PT_KDSR1_ADR
EXEC_PROCESS#	PT_KDSR2_ADR
INACTIVE	READY
IPC_QUOTA	ROOT_ASTE#
MEM_QUOTA	SDR_READ_ACCESS
MOV	SEG_FLAGS
NEG	SEG#_MAX

3.2.20.5 Limitations

STARTP returns ERR_FLAG if PS_CURRENT_PROCESS does not equal EXEC_PROCESS# or if PT_FLAGS (PROCESS#) and PT_FLAGS_MASK are not INACTIVE. Otherwise, RC = OK_FLAG.

3.2.20.6 Listing

```
DATA STARTP(USER, PROJECT, CLASS, CAT, PROCESS#, PROC_OFFSET) RETURNS (RC);




PROGRAM STARTP;
     DECLARE
          WORD (I, PDD_SEG#, PD_SEG#, CD_SEG#, SS_SEG#, KS_ASTE#, PROC_SEG#, DUMMY);

     /*    ONLY EXECUTIVE CAN CALL THIS FUNCTION                                    */

     IF PS_CURRENT_PROCESS ¬= EXEC_PROCESS#;
       THEN:
....      RETURN WITH ERR_FLAG;
     END;
```

```
/*     PROCESS MUST BE FREE                                                    */

IF (PT_FLAGS(PROCESS#) & PT_FLAGS_MASK) ¬= INACTIVE;
  THEN:
....      RETURN WITH ERR_FLAG;
END;

/*     MAKE "PARTIAL" SWITCH TO USER PROCESS                                   */

LSD(PT_PS_ASTE#(PROCESS#), PS_KSR_ADR, SDR_WRITE_ACCESS);

/*     INITIALIZE PS                                                           */

PS_CURRENT_PROCESS := PROCESS#;

/*     NEED MASK + NOTMASK                                                     */

INLINE(MOV, PROCESS#, 0, 3);
INLINE(DEC, 0, 3);
INLINE(NEG, 0, 3);
INLINE(MOV, 2, 7, 0, 0, "4000");
INLINE(MOV, 2, 7, 0, 1, "BFFF");
INLINE(ASHROR3);
INLINE(ASHR1R3);
INLINE(MOV, 0, 0, PS_PROCESS_MASK);
INLINE(MOV, 0, 1, PS_PROCESS_NOTMASK);
PS_USER_ID := USER;
PS_PROJECT_ID := PROJECT;
PS_CUR_CLASS := CLASS;
PT_CUR_CLASS(PROCESS#) := CLASS;
PS_CUR_CAT := CAT;
PT_CUR_CAT(PROCESS#) := CAT;
PS_MEM_QUOTA := MEM_QUOTA;
PT_IPC_QUOTA(PROCESS#) := IPC_QUOTA;

DO I := 0 TO 15;
     PS_SAR(I) := 0;
     PS_SDR(I) := 0;
END;

DO I := 0 TO SEG#_MAX;
     PS_SEG(I) := ((I + 1) | SEG_FLAG);
END;
PS_SEG(SEG#_MAX) := SEG_FLAG;

/*     PUT ROOT INTO "B"                                                       */

PS_SEG(0) := PS_SEG(1);
PS_SEG(1) := ROOT_ASTE#;
AST_CPL(ROOT_ASTE#) := (AST_CPL(ROOT_ASTE#) | PS_PROCESS_MASK);

/*     GAIN ACCESS TO STACKS                                                   */

/*     FIRST PDD                                                               */

PDD_SEG# := GETR(ROOT_ASTE#, PDD_OFFSET);

/*     NEXT EXEC'S PD                                                          */

PD_SEG# := GETR(PS_SEG(PDD_SEG#), EXEC_PROCESS#);

/*     NOW STACKS - FIRST S STACK                                              */

SS_SEG# := GETW(PS_SEG(PD_SEG#), PROCESS#);
DUMMY := ENABLE(PS_SEG(SS_SEG#), 0);
```

```
/*    K STACK IS AWKWARD BECAUSE GETW MUST FAIL                        */
/*    HOWEVER EXEC HAS DONE THE GETW AND AN ENABLE                     */

LSD(PS_SEG(PD_SEG#), DIR_KSR_ADR, SDP_READ_ACCESS);
KS_ASTE# := HASH(DIR_DISK(PROCESS#_MAX + PROCESS#));

IF KS_ASTE# = 0;
  THEN: INLINE(0);
END;

LSD(KS_ASTE#, PT_KSDR2_ADR + PROCESS# + PROCESS#, SDR_WRITE_ACCESS);
AST_DES_COUNT(KS_ASTE#) := AST_DES_COUNT(KS_ASTE#) + 1;
PT_KS_ASTE#(PROCESS#) := KS_ASTE#;

/*    CLEAN UP A BIT                                                   */

DCONNECT(PD_SEG#, PS_SEG(PD_SEG#));
DCONNECT(PDD_SEG#, PS_SEG(PDD_SEG#));

/*    NOW FOR INITIAL PROC                                             */

CD_SEG# := GETR(ROOT_ASTE#, CD_OFFSET);
PROC_SEG# := GETR(PS_SEG(CD_SEG#), PROC_OFFSET);
DUMMY := ENABLE(PS_SEG(PROC_SEG#), 2);
DCONNECT(CD_SEG#, PS_SEG(CD_SEG#));

/*    SWITCH BACK TO EXECUTIVE                                         */

LSD(PT_PS_ASTE#(EXEC_PROCESS#), PS_KSR_ADR, SDR_WRITE_ACCESS);
LSD(PT_PS_ASTP#(PROCESS#), PT_KSDR1_ADR + PROCESS# + PROCESS#, SDR_WRITE_ACCESS);
PT_R5(PROCESS#) := 0;

PT_FLAGS(PROCESS#) := READY;
PT_IPC_QUEUE_HEAD(PROCESS#) := 0;
RC := OK_FLAG;
```

3.2.21 Change Object (CHANGEO)

The Change Object CPC, CHANGEO is a user level external SKCPP function that is called by one user level external program, the executive process, with the parameters process#, aste#, offset, class and cat. CHANGEO calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.21.1 Description

CHANGEO alters the classification and category of a data segment according to the specification of a trusted subject. First, it checks that the calling process is indeed trusted. If PS_CURRENT_PROCESS is not EXEC_PROCESS#, ERR_FLAG is returned. Also, if WRITEDIR(ASTE#) does not return OK_FLAG, CHANGEO returns ERR_FLAG: the process must have write access to the parent segment, which must be a directory. The implementation requirement is that the segment whose attributes are to be changed must exist; if DIR_SIZE(OFFSET) is zero, it returns ERR_FLAG. Next, CHANGEO must insure that the segment is not in the WS of any process. It sets OASTE# to the aste# which HASH associates with DIR_DISK(OFFSET), the disk address of the segment. If OASTE# is Ø, the segment is inactive and by

definition is not in the WS of any process. Also if AST_CPL logical and WIRED_DOWN_NOTMASK - the wired down bit shares a word with the CPL equals Ø, it is not connected to any process. If neither of these conditions holds, some process has the segment in its WS, so the segments attributes cannot be changed: ERR_FLAG is returned. Also if DIR_TYPE(OFFSET) and DIR_TYPE_MASK equals DIR_TYPE_DIRECTORY, CHANGEO returns with ERR_FLAG. Finally, the compatibility rule that security levels must be nondecreasing as one moves down in the hierarchy is implemented. If CLASS is less than AST_CLASS(ASTE#) logical and AST_CLASS_MASK, the classification of the parent, or if the CAT set does not equal logical or AST_CAT(ASTE#), CHANGEO returns with ERR_FLAG.

Otherwise, checking is complete. DIR_CLASS(OFFSET) is reset to the logical or of CLASS and the logical and of DIR_CLASS(OFFSET) and DIR_CLASS_NOTMASK, changing the class without affecting the type and status bits. DIR_CAT(OFFSET) is assigned the value of CAT. If the segment is active, that is, if OASTE# is not equal to zero, ASTE_CLASS (OASTE#) and AST_CAT(OASTE#) must be changed also; they are set equal to DIR_CLASS(OFFSET) and DIR_CAT(OFFSET). RC is set to OK_FLAG and CHANGEO returns.

```
Function   CHANGEO
Parameters:  CHANGEO(process#,aste#,offset,class,cat);
Effect:
IF   (PS_TYPE(process#) ≠ TRUSTED)|
     not AST_WAL(aste#,process#)|
     (Ast_TYPE(aste#) ≠ DIRECTORY|
     (DIR_SIZE(aste#,offset) = 0|
     (HASH(DIR_DISK(aste#,offset#)) ≠ 0|
     (DIR_TYPE(aste#,offset#) ≠ DIRECTORY|
     (cat ⊉ AST_CAT(aste#))|
     (class < AST_CLASS(aste#);
THEN:  RC(process#) = NO;
ELSE:  DIR_CLASS(aste#) = class;
     DIR_CAT(aste#) = cat
     RC(process#) YES;
END;
```

3.2.21.2 N/A

3.2.21.3 <u>Interfaces</u>

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
PCHECK	WRITEDIR HASH

3.2.21.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function CHANGEO. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Local Paremeters	Local References
PS_CURRENT_PROCESS	ASTE#	OASTE#
DIR_SIZE	OFFSET	RC
DIR_TYPE	CLASS	
DIR_CLASS	CAT	
DIR_CAT		
AST_CPL		
AST_CLASS		
AST_CAT		

Constants

AST_CLASS_MASK	ERR_FLAG
DIR_CLASS_NOTMASK	EXEC_PROCESS#
DIR_TYPE	INACTIVE
DIR_TYPE_DIRECTORY	OK_FLAG
DIR_TYPE_MASK	WIRED_DOWN_MASK

3.2.21.5 Limitations

CHANGEO returns ERR_FLAG if PS_CURRENT_PROCESS does not equal EXEC_PROCESS#, if the intended parent segment is a directory to which the process does not have write access, if the segment does not exist in main memory, if the segment is in the WS of some process, if the segment is a directory, or if the category set is less than classification of the parent segment. Otherwise, RC = OK_FLAG.

3.2.21.6 Listing

```
DATA CHANGEO(ASTE*, OFFSET, CLASS, CAT) RETURNS (RC);
```

```
PROGRAM CHANGEO;
     DECLARE
          WORD (OASTE#);

     /*    SECURITY CHECKS                                                                  */

     /*    ONLY TRUSTED SUBJECTS CAN USE THIS FUNCTION                                      */

     IF PS_CURRENT_PROCESS ¬= EXEC_PROCESS#;
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     /*    INTERPRETIVE DIRECTORY WRITE CHECK                                               */

     IF WRITEDIR(ASTE#) ¬= OK_FLAG;
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     IF DIR_SIZE(OFFSET) = 0;
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     /*    IF OBJECT IS INACTIVE THEN NO NEED TO DO SECURITY & * -PROPERTY CHECK            */

     OASTE# := HASH(DIR_DISK(OFFSET));

     IF (OASTE# ¬= 0) & ((AST_CPL(OASTE#) & WIRED_DOWN_NOTMASK) ¬= 0);
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     /*    COMPATABILITY CHECK SIMPLIFIED IF OBJECT IS NOT A DIRECTORY                      */

     IF (DIR_TYPE(OFFSET) & DIR_TYPE_MASK) = DIR_TYPE_DIRECTORY;
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     IF CLASS < (AST_CLASS(ASTE#) & AST_CLASS_MASK);
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     IF CAT ¬= (AST_CAT(ASTE#) | CAT);
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     /*    CHECKING COMPLETE - PERFORM STATE CHANGE                                         */

     DIR_CLASS(OFFSET) := ((DIR_CLASS(OFFSET) & DIR_CLASS_NOTMASK) | CLASS);
     DIR_CAT(OFFSET) := CAT;

     /*    IF WIRED DOWN CHANGE ASTE CLASS AND CAT ALSO                                     */

     IF OASTE# ¬= 0;
       THEN: AST_CLASS(OASTE#) := DIR_CLASS(OFFSET);
             AST_CAT(OASTE#) := CAT;
     END;

     RC := OK_FLAG;
```

3.2.22 Initialize Hierarchy (INITH)

The Initialize Hierarchy CPC, INITH, is a user level external SKCPP function that is called by only one user level program, the executive process. INITH calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.22.1 Description

INITH allows the executive process to set up the directory structure at system initialization: it copies the attributes of a specified ASTE into a specified directory entry.

It checks that the calling process is trusted and that the directory aste# parameter supplied identifies a directory to which the process has write access. If PS_CURRENT_PROCESS does not equal EXEC_PROCESS# or if WRITEDIR (DASTE#) does not return ERR_FLAG, INITH returns with ERR_FLAG. It also requires that the directory entry specified be empty: if DIR_SIZE(OFFSET) is not zero, it returns with ERR_FLAG.

If the security and implementation requirements have been satisfied, it performs the state change. DIR_CLASS(OFFSET), DIR_CAT(OFFSET), DIR_DISK(OFFSET) and DIR_SIZE(OFFSET) are assigned, respectively, the values of AST_CLASS(ASTE#), AST_DISK, and AST_SIZE(ASTE#). Setting DIR_ACL_HEAD(OFFSET) to zero empties the access control list and completes the procedure. INITH returns with RC set equal to OK_FLAG.

```
Function:  INITH
Parameters:  INITH(process#,daste#,offset#,aste#)
Effect:
IF  (PS_TYPE(process#) ≠ TRUSTED|
    not AST_WAL(daste#)|
    (AST_TYPE(daste#) ≠ DIRECTORY|
    (DIR_SIZE(daste#,offset#) ≠ 0;
THEN:  RC(process#) = NO;
ELSE:  DIR_CLASS(daste#,offset#) = AST_CLASS(aste#);
   DIR_CAT(daste#,offset#) = AST_CAT(aste#);
   DIR_DISK(daste#,offset#) = AST_DISK(aste#);
   DIR_SIZE(daste#,offset#) = AST_SIZE(aste#);
   DIR_ACL_HEAD(daste#,offset#) = 0;
   RC(process#) = YES
END;
```

3.2.22.2 N/A

3.2.22.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
PCHECK	WRITEDIR

3.2.22.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function INITH. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Functional Parameters	Local References
PS_CURRENT_PROCESS	DASTE#	RC
DIR_CLASS	OFFSET	
DIR_CAT	ASTE#	
DIR_SIZE		
DIR_DISK		

Constants

ERR_FLAG
EXEC_PROCESS#
OK_FLAG

3.2.22.5 Limitations

INITH returns ERR_FLAG if PS_CURRENT_PROCESS does not equal EXEC_PROCESS#, if the directory entry specified is not empty, or if the intended parent segment is a directory to which the process does not have write access. Otherwise, RC=OK_FLAG.

3.2.22.6 Listing

```
DATA INITH(DASTE#, OFFSET, ASTE#) RETURNS (RC);
```

```
PROGRAM INITH;
                                                                              */
     /*    SECURITY CHECKS
                                                                              */
     /*    ONLY TRUSTED SUBJECTS CAN USE THIS FUNCTION

     IF PS_CURRENT_PROCESS ¬= EXEC_PROCESS#;
       THEN:
....       RETURN WITH ERR_FLAG;
     END;
                                                                              */
     /*    INTERPRETIVE DIRECTORY WRITE CHECK

     IF WRITEDIR(DASTE#) ¬= OK_FLAG;
       THEN:
....       RETURN WITH ERR_FLAG;
     END;
                                                                              */
     /*    IMPLEMENTATION CHECKS

     IF DIR_SIZE(OFFSET) ¬= 0;
       THEN:
....       RETURN WITH ERR_FLAG;
     END;
                                                                              */
     /*    PERFORM STATE CHANGE - COPY ASTE ATTRIBUTES INTO DIRECTORY ENTRY

     DIR_CLASS(OFFSET) := AST_CLASS(ASTE#);
     DIR_CAT(OFFSET) := AST_CAT(ASTE#);
     DIR_DISK(OFFSET) := AST_DISK(ASTE#);
     DIR_SIZE(OFFSET) := AST_SIZE(ASTE#);
     DIR_ACL_HEAD(OFFSET) := 0;
     RC := OK_FLAG;
```

3.2.23 Get Directory (GETDIR)

The GET Directory CPC, GETDIR, is a kernel level internal SKCPP function that is called by a user level external function. GETDIR calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.23.1 Description

GETDIR insures that a directory segment is swapped into main memory and can be accessed. If AST_ADR(ASTE#) is zero, the ASTE# supplied identifies a segment not in main memory; SWAPIN is called to swap the segment in. LSD is then called to load the segment descriptors in the register at DIR_KSR_ADR to provide write access to the segment.

```
Function:  GETDIR
Parameters:  GETDIR(aste#)
IF  AST_ADR - 0
THEN:  SWAPIN;
LSD;
AST_CHANGED;
ELSE:  LSD;
   AST_CHANGED:
END;
```

3.2.23.2 N/A

3.2.23.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
DELETE	SWAPIN
	LSD

3.2.23.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function GETDIR. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
AST_ADR	ASTE#	None

Constants

DIR_KSR_ADR
SDR_WRITE_ACCESS

3.2.23.5 Limitations

None.

3.2.23.6 Listing

```
DATA GETDIR(ASTE#);
```

```
PROGRAM GETDIR;

    /*    LOAD SEGMENT DESCRIPTOR FOR DIRECTORY                                */

    IF AST_ADR(ASTE#) = 0;
      THEN: SWAPIN(ASTE#);
    END;

    LSD(ASTE#, DIR_KSR_ADR, SDR_WRITE_ACCESS);
```

3.2.24 Write Directory (WRITEDIR)

The Write Directory CPC, WRITEDIR, is a kernel level internal SKCPP that is called by user level external functions. WRITEDIR calls kernel level internal functions. It is written in Project SUE System Language.

3.2.24.1 Description

WRITEDIR makes security and implementation checks, and if constraints are met, provides access to a specified directory. It returns ERR_FLAG if AST_TYPE(ASTE#) logical and AST_TYPE_MASK does not equal AST_TYPE_DIRECTORY: that is, if the segment supplied is not a directory. If AST_WAL(ASTE#) logical and PS_PROCESS_MASK, which contains all Ø's except a 1 in the bit corresponding to the process#, is zero, it returns with ERR_FLAG.

If these requirements are satisifed, it makes certain that the segment is in main memory. If AST_ASR(AST#), which holds the main memory address of the segment, is zero, WRITEDIR calls SWAPIN to swap the segment into main memory. Next, it calls LSD, which loads the descriptors of the directory in the register located at DIR_KSR_ADR with access mode SDR_WRITE_ACCESS. Since write access to the directory has been gained, the directory will be changed, so, to insure that the segment will be copied onto the disk when it is swapped out, the change bit is set. WRITEDIR lets AST_CHANGE(ASTE#) equal AST_CHANGE(ASTE#) logical or AST_CHANGED. Setting RC equal to OK_FLAG, it returns.

Function: WRITEDIR
Paremeters: WRITEDIR(aste#)
Effect:
IF <u>not</u> (AST_TYPE(aste#) ≠ DIRECTORY)|
    (AST_WAL(aste#) ≠ 0)

```
THEN:  RC(TCP) = NO;
ELSE:  RC(TCP) = YES
   IF  AST_ADR = 0;
   THEN:  SWAPIN:
      LSD;
      AST_CHANGED;
   ELSE:  LSD;
      AST_CHANGED;
   END;
END:
```

3.2.24.2. N/A

3.2.24.3. Interfaces

Refer to Figure 6, Function Call Matrix, in Paragraph 3.4.

Called By	Calls
CREATE	SWAPIN
DELETE	LSD
GIVE	
RESCIND	
CHANGO	
INITH	

3.2.24.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function WRITEDIR. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
PS_PROCESS_MASK	ASTE#	RC
AST_ADR		
AST_TYPE		
AST_WAL		
AST_CHANGE		

Constants

AST_CHANGE

Constants cont.

AST_TYPE_DIRECTORY
AST_KSR_ADR
ERR_FLAG
OK_FLAG
SDR_WRITE_ACCESS

3.2.24.5 Limitations

WRITEDIR returns ERR_FLAG if the segment supplied is not a directory, or if the process does not have write access to the directory. Otherwise, RC = OK_FLAG.

3.2.24.6 Listing

```
DATA WRITEDIR(ASTE#) RETURNS (RC);

PROGRAM WRITEDIR;

     /*    CHECKS THAT SPECIFIED SEGMENT IS A DIRECTORY AND THAT IT IS IN CURRENT       *
      * PROCESS'S B IN WRITE MODE                                                       */

     IF (AST_TYPE(ASTE#) & AST_TYPE_MASK) ¬= AST_TYPE_DIRECTORY;
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     IF (AST_WAL(ASTE#) & PS_PROCESS_MASK) = 0;
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     IF AST_ADR(ASTE#) = 0;
       THEN: SWAPIN(ASTE#);
     END;

     /*    GAIN ACCESS TO THE DIRECTORY                                                 */

     LSD(ASTE#, DIR_KSR_ADR, SDR_WRITE_ACCESS);

     /*    DIRECTORY WILL BE CHANGED - SET CHANGE BIT                                   */

     AST_CHANGE(ASTE#) := (AST_CHANGE(ASTE#) | AST_CHANGED);
     RC := OK_FLAG;
```

3.2.25 Delete Segment (DELETSEG)

The Delete Segment CPC, DELETSEG, is a kernel level internal SKCPP function that is called by a user level external function. DELETSEG calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.25.1 Description

DELETSEG deletes a data segment or an empty directory segment. First, it removes any elements that may be on the ACL for the segment. It sets INDEX equal to DIR_ACL_HEAD(OFFSER). If INDEX is not zero, there are some ACL elements to be removed. Until ACL_CHAIN(INDEX), which holds the acle# of the next element in the list, equals zero, INDEX is reset to ACL_CHAIN(INDEX). ACL_CHAIN (INDEX) is then set to ACL_CHAIN(Ø), linking the end of the ACL to the head of the free chain, and ACL_CHAIN(Ø), the pointer to the free acle chain, is set to DIR_ACL_HEAD(OFFSET). Setting DIR_ACL_HEAD(OFFSET) to Ø completes the transfer of the ACL to the free chain.

DELETSEG then calls SOADD, which removes the segment from the WS of any process whose access rights have been rescinded. Since the segment's ACL has just been emptied, SOADD removes the segment from all WS's.

If the segment to be deleted is active (aged, now), it must be deactivated. When HASH is called with a parameter of DIR_DISK (OFFSET), which contains the disk address of the segment, it returns the aste# of the segment. This value is assigned to OASTE#. If OASTE# is non-zero, the segment is active and eligible for deactivation. The change and status bits are zeroed by letting AST_CHANGE (OASTE#) equal AST_CHAIN(OASTE#) logical and AST_UNCHANGED_MASK and AST_STATUS(OASTE#) equal AST_STATUS(OASTE#) logical and AST_STATUS_NOTMASK. DEACT is then called to swap the segment out of main memory, if necessary, and move its aste from the list of segments eligible for deactivation to the list of free aste's.

DELETSEG then calls DFREE to free the disk space occupied by the segment. It sets DIR_DISK(OFFSET) and DIR_SIZE(OFFSET) to zero to mark the directory entry free. Finally, DELETSEG sets the change bit in the parent directory so it will be copied onto the disk when swapped out of main memory: AST_CHANGE(ASTE#) is set equal to AST_CHANGE(ASTE#) logical or AST_CHANGED. It then returns control to the calling program.

```
Function:  DELETSEG
Parameters:  DELETSEG(aste#, offset)
Effect:
IF  'DIR_ACL_HEAD'(aste#, offset) ≠ 0;
THEN:  Let acle# = FINDEND(aste#, 'DIR_ACL_HEAD'(aste#, offset);
   ACL_CHAIN(aste#, acle#) = 'ACL_CHAIN'(aste#, 0);
```

```
      ACL_CHAIN(aste#, 0) = 'DIR_ACL_HEAD'(aste#, offset);
      SOADD(aste#, offset);
   END;
   IF  HASH('DIR_DISK'(aste#, offset)) ≠ 0);
   THEN:  DEACTIVATE(HASH('DIR_DISK'(aste#, offset)));
   END;
   DISK_FREE('DIR_DISK'(aste#, offset, 'DIR_SIZE'(aste#,
            offset)));
   DIR_SIZE(aste#, offset) = 0;
```

3.2.25.2 N/A

3.2.25.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
DELETE	SOADD
	DFREE
	DEACT
	HASH

3.2.25.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function DELETSEG. For data base references refer to Figure 5, Data Base References Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
ACL_CHAIN	ASTE#	OASTE#
ACL_CHANGE		INDEX
DIR_ACL_HEAD		
DIR_DISK		
DIR_SIZE		

Constants

AST_CHANGE
AST_STATUS_NOTMASK
AST_UNCHANGED_MASK
BMT_SIZE2_ADR
ERR_FLAG

3.2.25.5 Limitations

None

3.2.25.6 Listing

```
DATA DELETSEG(ASTE#, OFFSET);
     DECLARE
          PROCEDURE ACCEPTS (WORD, WORD) (DFREE);


PROGRAM DELETSEG;
     DECLARE
          WORD (INDEX, OASTE#);

     /*   REMOVE ANY ELEMENTS THAT MAY BE ON ACL CHAIN                         */

     INDEX := DIR_ACL_HEAD(OFFSET);

     IF INDEX ¬= 0;
       THEN:

          CYCLE
          .... EXIT WHEN ACL_CHAIN(INDEX) = 0;
               INDEX := ACL_CHAIN(INDEX);
          END;

          ACL_CHAIN(INDEX) := ACL_CHAIN(0);
          ACL_CHAIN(0) := DIR_ACL_HEAD(OFFSET);
          DIR_ACL_HEAD(OFFSET) := 0;
     END;

     /*   NOW BUMP EVERYBODY OFF                                               */

     SOADD(ASTE#, OFFSET);

     /*   DEACTIVATE IF ASTE# OF OFFSET (OASTE#) IS AGED                       */

     OASTE# := HASH(DIR_DISK(OFFSET));

     IF OASTE# ¬= 0;
       THEN:

          /*   SET CHANGE BIT TO UNCHANGED AND STATUS BIT TO INITIALIZED       */

          AST_CHANGE(OASTE#) := (AST_CHANGE(OASTE#) & AST_UNCHANGED_MASK);
          AST_STATUS(OASTE#) := (AST_STATUS(OASTE#) & AST_STATUS_NOTMASK);
          DEACT(OASTE#);
     END;

     /*   FREE UP RESOURCES - DISK SPACE AND DIRECTORY ENTRY                   */

     DFREE(DIR_DISK(OFFSET), BMT_SIZE2_ADR);
     DIR_DISK(OFFSET) := 0;
     DIR_SIZE(OFFSET) := 0;

     /*   SET CHANGE BIT IN PARENT                                             */

     AST_CHANGE(ASTE#) := (AST_CHANGE(ASTE#) | AST_CHANGED);
```

3.2.26 Connect (CONNECT)

The Connect CPC, CONNECT, is a kernel level internal SKCPP function that is called by user level external functions. CONNECT calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.26.1 Description

CONNECT connects a process to a segment, putting the segment in the WS of the process. It is subject to two implementation constraints. It sets SEG# to PS_SEG(Ø) logical and SEG_MASK. If SEG# equals zero, there are no free segment numbers, and CONNECT returns with ERR_FLAG. It then insures that the segment is active. It calls HASH which returns the AST entry number associated with DIR_DISK(OFFSET), the disk address of the segment, and assigns this value to ASTE#. If ASTE# is zero, CONNECT must call ACT to activate the segment, and ASTE# is set to the value it returns. If ASTE# is non-zero, showing that the segment is already active, and AST_CPL (ASTE#) logical and PS_PROCESS_MASK is non-zero, the process is already connected to the segment so CONNECT returns with ERR_FLAG.

Otherwise, it can proceed with the connection. If the segment is eligible for deactivation it must be removed from the age chain. If AST_CPL(ASTE#) is Ø, this is the case. To find the segment's AST entry number in the age chain, it sets INDEX to Ø and NEXT to AST_AGE_CHAIN(Ø), which holds the head of the age chain. Then, until it has set NEXT to ASTE#, it resets INDEX and NEXT to AST_AGE_CHAIN(INDEX). Assigning to AST_AGE_CHAIN(INDEX) the value of AST_AGE_CHAIN(ASTE#) and setting AST_AGE_CHAIN(ASTE#) to Ø removes the segment from the chain.

CONNECT now performs the actual connection by resetting AST_CPL(ASTE#) to AST_CPL(ASTE#) logical or PS_PROCESS_MASK. Since PS_PROCESS_MASK consists of all Ø's except for a 1 in the bit corresponding to the process#, this sets the CPL bit for the process. If the MODE is WRITE$READ$EXECUTE_ACCESS, the WAL bit for the process must also be set. CONNECT accomplishes this by resetting AST_WAL (ASTE#) to AST_WAL(ASTE#) logical or PS_PROCESS_MASK.

To complete the procedure, PS_SEG(Ø) is reset to PS_SEG(SEG#), which removes SEG# from the free segment chain, and PS_SEG(SEG#) is set to ASTE#. This allows segment numbers to be mapped onto AST entry numbers, which is necessary because ASTE numbers are system-wide variables to whose values users can not have access. CONNECT returns with RC set to the SEG# with which the user can subsequently refer to the segment.

```
Function:  CONNECT
Parameters:  CONNECT(process#, daste#, entry#, mode)
Effect:
IF  'PS_SEG'(process#, 0) = 0;
THEN:  RC(process#) = NO;
ELSE:  Let flag = 'HASH'(DIR_DISK(daste#, entry#));
   IF  (flag ≠ 0) &
        'AST_CPL'(flag, process#);
   THEN:  RC(process#) = NO;
   ELSE:
      IF  flag ≠ 0;
      THEN:  Let aste# = flag;
         IF  'AST_AGE' (aste#) = AGED;
         THEN:  UNAGE(aste#);
         END;
      ELSE:  ACTIVATE(daste#, entry#);
         Let aste# = HASH(DIR_DISK(daste#, entry#));
         UNAGED(aste#);
      END;
      AST_CPL(aste#, process#) = TRUE;
      IF  mode = WRITE;
      THEN:  AST_WAL(aste#, process) = TRUE;
      END;
      Let seg# = 'PS_SEG'(process#, seg#);
      PS_SEG(process#, 0) = 'PS_SEG'(process#, seg#);
      PS_SEG(process#, seg#) = aste#;
      PS_SEG_INUSE(process#, seg#) = TRUE;
      RC(process#) = YES, seg#;

      END;
END;
```

3.2.26.2 N/A

3.2.26.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
GETW	ACT
GETR	HASH

3.2.26.4 Data Organization

Listed below are Security Kernel data base references and constants used by function CONNECT. For data base references

refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
PS_SEG	DASTE#	INDEX
PS_PROCESS_MASK	OFFSET	NEXT
AST_CPC	MODE	SEG#
AST_AGE_CHAIN		ASTE#
AST_WAL		HASH_VAL
DIR_DISK		RC

Constants

ERR_FLAG
OFFSET_MAX
OFFSET_MIN
SEG_MASK
WRITE$READ$EXECUTE_ACCESS

3.2.26.5 Limitations

CONNECT returns EFF_FLAG if there are no free segment numbers or if the process is already connected to the segment. Otherwise, RC = SEG#.

3.2.26.6 Listing

```
DATA CONNECT(DASTE#, OFFSET, MODE) RETURNS (RC);


PROGRAM CONNECT;
     DECLARE
          WORD (INDEX, NEXT, SEG#, ASTE#, HASH_VAL);

     /*    FIND A FREE SEG#                                              */

     SEG# := (PS_SEG(0) & SEG_MASK);

     IF SEG# = 0;
       THEN:
....      RETURN WITH ERR_FLAG;
     END;

     /*    DETERMINE IF SEGMENT IS ACTIVE                                */

     ASTE# := HASH(DIR_DISK(OFFSET));
```

```
IF ASTE# = 0; /* THEN: MUST ACTIVATE */
  THEN: ASTE# := ACT(DASTE#, OFFSET);
  ELSE:
      IF (AST_CPL(ASTE#) & PS_PROCESS_MASK) ¬= 0;
        THEN:
....          RETURN WITH ERR_FLAG;
      END;

END;

/*    UNAGE IF NECESSARY                                          */

IF AST_CPL(ASTE#) = 0;
  THEN: INDEX := 0;

      CYCLE
           NEXT := AST_AGE_CHAIN(INDEX);
      .... EXIT WHEN NEXT = ASTE#;
           INDEX := NEXT;
      END;

      AST_AGE_CHAIN(INDEX) := AST_AGE_CHAIN(ASTE#);
      AST_AGE_CHAIN(ASTE#) := 0;
END;

/*    ADD THIS PROCESS TO CONNECTED PROCESS LIST                  */

AST_CPL(ASTE#) := (AST_CPL(ASTE#) | PS_PROCESS_MASK);

IF MODE = WRITE$READ$EXECUTE_ACCESS;
  THEN: AST_WAL(ASTE#) := (AST_WAL(ASTE#) | PS_PROCESS_MASK);
END;

/*    AND UPDATE PS_SEG                                           */

PS_SEG(0) := PS_SEG(SEG#);
PS_SEG(SEG#) := ASTE#;
RC := SEG#;
```

3.2.27 Search Out and Destroy Descriptors (SOADD)

The Search Out and Destroy Descriptors CPC, SOADD, is a kernel level internal SKCPP function that is called by both user level external functions and kernel level internal functions. SOADD calls both user level external functions and kernel level internal functions. It is written in Project SUE System Language.

3.2.27.1 Description

SOADD searches out and destroys descriptors for a segment in processes whose access rights have been restricted. It checks that the segment is active - if it is not, no processes are connected to it at all. It calls HASH to find the aste# associated with the segment's disk address, which is held in DIR_DISK(OFFSET). This value it assigns to ASTE#. Since HASH returns a zero if the segment is active, and since the CPL contains 1's in the bits corresponding to connected processes, if ASTE# and AST_CPL(ASTE#) are both non-zero, SOADD must proceed. Otherwise, no processes have descriptors for the segment and the call is ignored.

It then loops through all processes, PROCESS# going from PROCESS#_MIN to PROCESS#_MAX. If PT_FLAGS(PROCESS#) logical and PT_FLAGS_MASK (FLAGS and LINK entries share a byte) equal inactive, there is no need to check the process. If it is not INACTIVE, SOADD must ensure that it is not connected to the segment without adequate access rights. To find out, it needs access to the process's process segment (PS). Hence, LSD is called to load the descriptors of the process segment, identified by PT_PS_ASTE#(PROCESS#) into the kernel register 2 at PS_KSR_ADR.

SOADD then determines whether the process is connected to the segment, and if so, in what mode of access. If AST_CPL(ASTE#) logical and PS_PROCESS_MASK, which contains a single 1 in the bit corresponding to the PROCESS#, is non-zero, the CPL bit for the process is set. If it isn't, SOADD continues looping through the processes. If it is, SOADD checks if the process is also on the segment's write access list - if AST_WAL(ASTE#) logical and PS_PROCESS_MASK are zero, the process has read access only, and MODE is set to READ$EXECUTE_ACCESS.

SOADD then calls DSEARCH which searches the ACL of segment DASTE#, OFFSET to see if access MODE is still permitted. If DSEARCH returns OK_FLAG, the MODE of access is permitted; if DSEARCH returns ERR_FLAG, the MODE is no longer permitted, and SOADD destroys the descriptor. To do this, it must find the seg# of the segment: it loops through all the seg#'s from SEG#_MIN to SEG#_MAX until it finds the one which corresponds to the segment, the one for which PS_SEG(SEG#) equals ASTE#. SOADD then calls DCONNECT to release this segment from the process's WS,and continues its loop through the processes.

After it has checked the last process, PROCESS#_MAX, the kernel segmentation register 2 must be restored with the current process's process segment. SOADD calls LSD to load the descriptors of PT_PS_ASTE#(THE_CURRENT_PROCESS) into the register at PS_KSR_ADR and then returns.

```
Function:  SOADD
Parameters:  SOADD(daste#, offset)
Effect:
Let aste# = HASH(DIR_DISK(daste#, offset));
IF  aste# ≠ 0;
THEN:
   IF  (PROCESS#_MIN <= process# <= PROCESS#_MAX) &
       PT_FLAGS(process#) ≠ INACTIVE) &
       AST_CPL(aste#, process#);
```

```
THEN:
   IF  AST_WAL(aste#, process#);
   THEN:  Let mode = WRITE;
   ELSE:  Let mode = READ;
   END;
   IF not DSEARCH(process#, daste#
          'DIR_ACL_HEAD'(aste#, offset), mode);
   THEN:
      IF  (SEG#_MIX <= seg# <= SEG#_MAX) &
          ('PS_SEG'(process#, seg#) = aste#);
      THEN:  DCONNECT(process#, aste#, seg#);
      END;
   END;
   END;
END;
```

3.2.27.2 N/A

3.2.27.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
GIVE	DCONNECT
RESCIND	DSEARCH
DELETSEG	HASH
	LSD

3.2.27.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function SOADD. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
PS_PROCESS_MASK	DASTE#	ASTE#
PS_SEG	OFFSET	PROCESS#

Global References	Function Parameters	Local References
PT_FLAGS		MODE
AST_CPL		REG#
AST_WAL		SEG#
DIR_DISK		
THE_CURRENT_PROCESS		

Constants

ERR_FLAG	READ$EXECUTE_ACCESS
INACTIVE	SDR_WRITE_ACCESS
PROCESS#_MAX	SEG#_MAX
PROCESS#_MIN	SEG#_MIN
PS_KSR_ADR	WRITE$READ$EXECUTE_ACCESS
PT_FLAGS_MASK	

3.2.27.5 Limitations

None.

3.2.27.6 Listing

```
DATA SOADD(DASTE#, OFFSET);


PROGRAM SOADD;
    DECLARE
        WORD (ASTE#, PROCESS#, MODE, REG#, SEG#, DUMMY);

    /*    DETERMINE IF SEGMENT TO WHICH ACCESS HAS BEEN RESCINDED IS ACTIVE           */

    ASTE# := HASH(DIR_DISK(OFFSET));

    IF (ASTE# ¬= 0) & (AST_CPL(ASTE#) ¬= 0);
      THEN:

        /*    LOOP THROUGH ALL PROCESS'S                                              */

        DO PROCESS# := PROCESS#_MIN TO PROCESS#_MAX;

            IF (PT_FLAGS(PROCESS#) & PT_FLAGS_MASK) ¬= INACTIVE;
              THEN: LSD(PT_PS_ASTE#(PROCESS#), PS_KSR_ADR, SDR_WRITE_ACCESS);

                /*    IS THIS PROCESS CONNECTED TO THE SEGMENT?                       */

                IF (AST_CPL(ASTE#) & PS_PROCESS_MASK) ¬= 0;
                  THEN: /* YES - DETERMINE MODE OF ACCESS */

                    IF (AST_WAL(ASTE#) & PS_PROCESS_MASK) = 0;
                      THEN: MODE := READ$EXECUTE_ACCESS;
                      ELSE: MODE := WRITE$READ$EXECUTE_ACCESS;
                    END;
```

```
                    /*    DOES PROCESS STILL HAVE ACCESS RIGHTS?                 */

                    IF DSEARCH(DASTE#, OFFSET, MODE) = ERR_FLAG;
                      THEN:

                        DO SEG# := SEG#_MIN TO SEG#_MAX;

                            IF PS_SEG(SEG#) = ASTE#;
                              THEN: DCONNECT(SEG#, ASTE#);
                        ....          EXIT;
                            END;

                        END;

                    END;

                END;

            END;

        END;

        /*    RESTORE KSR2                                                   */

        LSD(PT_PS_ASTE#(THE_CURRENT_PROCESS), PS_KSR_ADR, SDR_WRITE_ACCESS);
    END;
```

3.2.28 Directory Search (DSEARCH)

The Directory Search CPC, DSEARCH, is a kernel level internal SKCPP function that is called by both a kernel level internal function and user level external functions. DSEARCH calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.28.1 Description

DSEARCH determines whether the mode of access is permitted for the calling process. It makes sure the directory aste# supplied does identify a directory, returning the ERR_FLAG if AST_TYPE(ASTE#) logical and AST_TYPE_MASK (type, class, status, change, and unlock entries share a byte) does not equal AST_TYPE_DIRECTORY. It then gains access to the directory. If AST_ASR(ASTE#), which holds the main memory address of the segment, equals zero, DSEARCH must call SWAPIN to swap the directory into main memory. It calls LSD to load the direcrory's descriptors in kernel segmentation register 3 at DIR_KSR_ADR.

It then searches for an ACL element concerning the current process, beginning at the head of the ACL, with INDEX set to DIR_ACL_HEAD(OFFSET). It commences a cycle to find the appropriate ACL element: if INDEX equals zero, the end of the ACL has been reached without finding an element which gives the process access; DSEARCH returns with ERR_FLAG. It sets USER to ACL_USER(INDEX) logical and ACL_USER_MASK and it sets PROJECT to ACL_PROJECT(INDEX).

When USER equals ALL_USERS or USER equals PS_USER_ID and PROJECT equals ALL_PROJECTS or PROJECT equals PS_PROJECT_ID, the cycle is exited; otherwise, INDEX is reset to ACL_CHAIN(INDEX), the next ACL element number and the cycle continued.

It now uses the ACL element it has found to test whether the requested MODE is permitted. If ACL_MODE(INDEX) logical and ACL_MODE_MASK (mode and user share a word) equals NO_ACCESS, DSEARCH returns with ERR_FLAG. Also, if requested MODE is WRITE$READ$EXECUTE_ACCESS and ACL_MODE(INDEX logical and ACL_MODE_MASK is not WRITE$READ$EXECUTE_ACCESS, ERR_FLAG is returned. Otherwise, DSEARCH returns with RC set to OK_FLAG.

```
Function:  DSEARCH
Parameters:  DSEARCH(process#, aste#, acle#, mode)
Value:
IF  acle# ≠ 0;
THEN:
   IF  ((ACL_USER(aste#, acle#) = ALL_USERS)|
        (ACL_USER(aste#, acle#) = PS_USER_ID(process#)) &
        ((ACL_PROJECT(aste#, acle#) = ALL_PROJECTS)|
        (ACL_PROJECT(aste#, acle#) = PS_PROJECT_ID(process#));
   THEN:
      IF  ACL_MODE(aste#, acle#) = NO;
      THEN:  FALSE;
      ELSE:
         IF  (mode = WRITE) &
             (ACL_MODE(aste#, acle#)  ≠ WRITE);
         THEN:  FALSE;
         ESLE:  TRUE;
         END;
      END;
   ELSE:  DSEARCH(process#, aste#, ACL_CHAIN(aste#, acle#),
          mode);
   END;
ELSE:  FALSE
END;
```

3.2.28.2 N/A

3.2.28.3 Interfaces

Refer to Figure 6, Function Call Matrix, inparagraph 3.4.

Called By	Calls
GETW	SWAPIN

Called By	Calls
GETR	LSD
SOADD	

3.2.28.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function DSEARCH. For Data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
AST_TYPE	DASTE#	RC
AST_ADR	OFFSET	
DIR_ACL_HEAD	ASTE#	
ACL_USER		
ACL_PROJECT		
ACL_CHAIN		
ACL_MODE		
PS_USER_ID		
PS_PROJECT_ID		

Constants

ACL_MODE_MASK	DIR_KSR_ADR
ACL_USER_MASK	EFF_FLAG
ALL_PROJECTS	NO_ACCESS
ALL_USERS	OK_FLAG
AST_TYPE_DIRECTORY	SDR_WRITE_ACCESS
AST_TYPE_MASK	WRITE$READ$EXECUTE_ACCESS

3.2.28.5 Limitations

DSEARCH returns ERR_FLAG if the aste# supplied does not identify a directory, if the end of the ACL has been reached without finding an element that gives the process access, or if the access mode is not permitted. Otherwise, RC = OK_FLAG.

3.2.28.6 Listing

```
DATA DSEARCH(ASTE#, OFFSET, MODE) RETURNS (RC);
```

```
PROGRAM DSEARCH;
     DECLARE
          WORD (INDEX, USER, PROJECT);

     IF (AST_TYPE(ASTE#) & AST_TYPE_MASK) ¬= AST_TYPE_DIRECTORY;
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     /*    GAIN ACCESS TO DIRECTORY                                                  */

     IF AST_ADR(ASTE#) = 0;
       THEN: SWAPIN(ASTE#);
     END;

     LSD(ASTE#, DIR_KSR_ADR, SDR_WRITE_ACCESS);

     /*    NO NEED TO CHECK A USE BIT - ACL WILL BE EMPTY IF ENTRY IS NOT IN USE      */

     /*    SEARCH ACL FOR ELEMENT THAT GIVES CURRENT USER PERMISSION TO ACCESS        */

     INDEX := DIR_ACL_HEAD(OFFSET);

     CYCLE

          IF INDEX = 0;
            THEN:
....             RETURN WITH ERR_FLAG;
          END;

          USER := (ACL_USER(INDEX) & ACL_USER_MASK);
          PROJECT := ACL_PROJECT(INDEX);
     .... EXIT WHEN ((USER = ALL_USERS) | (USER = PS_USER_ID)) & ((PROJECT =
               PS_PROJECT_ID) | (PROJECT = ALL_PROJECTS));
          INDEX := ACL_CHAIN(INDEX);
     END;

     IF (ACL_MODE(INDEX) & ACL_MODE_MASK) = NO_ACCESS;
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     IF (MODE = WRITE$READ$EXECUTE_ACCESS) & ((ACL_MODE(INDEX) & ACL_MODE_MASK) ¬=
          WRITE$READ$EXECUTE_ACCESS);
       THEN:
....        RETURN WITH ERR_FLAG;
     END;

     RC := OK_FLAG;
```

3.2.29 Activate Segment (ACT)

The Activate Segment CPC, ACT is a kernel level internal SKCPP function that is called by a user level external functions. ACT calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.29.1 Description

ACT activates a segment, copying its directory entry into the ASTE and initializing the other fields of the ASTE. First, it allocates an AST entry. It checks if any are on the free chain: if AST_CHAIN(Ø) is zero, there are none. In this event it deactivates the ASTE which has been eligible for deactivation longest. It sets I to Ø. It then repeatedly assigns NEXT the value of AST_AGE_CHAIN(I) and I the value of NEXT until AST_AGE_CHAIN(NEXT) equals Ø, marking the end of the age chain. NEXT now holds the aste# of the entry which was on the AGE chain longest. ACT calls DEACT to free this entry. Now that there is at least one aste# on the free chain, ASTE# can be set equal to AST_CHAIN(Ø), the aste# of the first element on the free chain. Assigning AST_CHAIN(Ø) the value of AST_CHAIN(ASTE#) removes the entry from the free chain. Since the segment is not connected to any process at activate time, it is marked eligible for deactivation by letting AST_AGE_CHAIN(ASTE#) equal AST_AGE_CHAIN(Ø) and AST_AGE_CHAIN(Ø) equal ASTE#.

Act then updates the HASH data base. It calls PREHASH to calculate the hash value HASH_VAL associated with DIR_DISK(OFFSET), the disk address of the segment. It then sets AST_CHAIN(ASTE#) to HASH_TABLE(HASH_VAL) and HASH_TABLE(HASH_VAL) to ASTE#. This adds the entry to the head of a chain of ASTE's whose entries disk addresses map onto the same hash value.

ACT is now ready to fill in the ASTE it has allocated. Since the segment is not yet connected to any processes and can't be swapped in yet, the main memory address, AST_ADR(ASTE#), the descriptor count, AST_DES_COUNT(ASTE#), and the CPL, AST_CPL(ASTE#) are all set to zero. The rest of the entry is copied from the directory entry. AST_CLASS(ASTE#) is set equal to DIR_CLASS(OFFSET) which sets the type and status as well as the classification. AST_CAT, AST_DISK, and AST_SIZE get DIR_CAT, DIR_DISK and DIR_SIZE, repectively.

Now, if the directory status bit was set to uninitialized, it must be reset. If DIR_STATUS(OFFSET) logical and DIR_STATUS_MASK equals DIR_UNINITIALIZED, DIR_STATUS(OFFSET) is reset to DIR_STATUS (OFFSET) logical and DIR_STATUS_NOTMASK (all 1's except for the status bit, so class and type information is not disturbed). In this case, the directory has been changed, so AST_CHANGE(DASTE#) must be reset to AST_CHANGE(DASTE#) logical or AST_CHANGED to insure that the directory will be copied onto disk when swapped out.

Finally, ACT initializes the segment's semaphore. SMFR_COUNT(ASTE#) is set to 1 and SMF_POINTER(ASTE#), the head of a chain of processes blocked on the semaphore, is set to Ø. ACT returns the ASTE# allocated to the segment.

```
Function:  ACTIVATE
Parameters:  ACTIVATE(daste#, offset)
Effect:
IF  'AST_CHAIN'(0) = 0;
THEN:  Let aste# = NEXTASTE('AST_AGE_CHAIN'(0));
   DEACTIVATE(aste#);
ELSE:  Let aste# = 'AST_CHAIN'(aste#);
   AST_CHAIN(0) = 'AST_CHAIN'(aste#);
END:
HASH(DIR_DISK(daste#, offset)) = aste#;
AST_ADR(aste#) = 0;
AST_LOCK(aste#) = UNLOCK;
AST_DES_COUNT(aste#) = 0;
IF  (PROCESS#_MIX <= process# <= PROCESS#_MAX);
THEN:  AST_CPL(aste#, process#) = FALSE;
   AST_WAL(aste#, process#) = FALSE;
END;
AST_TYPE(aste#) = DIR_TYPE(daste#, offset);
AST_STATUS(aste#) = 'DIR_STATUS'(daste#, offset);
AST_CLASS(aste#) = DIR_CLASS(daste#, offset);
AST_CAT(aste#) = DIR_CAT(daste#, offset);
AST_DISK(aste#) = DIR_DISK(daste#, offset);
AST_SIZE(aste#) = DIR_SIZE(daste#, offset):
IF  'DIR_STATUS'(daste#, offset) = UNINITIALIZED;
THEN:  DIR_STATUS(daste#, offset) = INITIALIZED;
END;
AGE(aste#);
```

3.2.29.2 N/A

3.2.29.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
DELETE	DEACT
CONNECT	PREHASH

3.2.29.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function ACT. For data base references

refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
AST_CHAIN	DASTE#	ASTE#
AST_AGE_CHAIN	OFFSET	HASH_VAL
AST_ADR		INDEX
AST_DES_COUNT		NEXT
AST_CPL		I
AST_CHANGE		RC
AST_CLASS		
AST_CAT		
AST_DISK		
AST_SIZE		
DIR_CLASS		
DIR_CAT		
DIR_DISK		
DIR_SIZE		
DIR_STATUS		
HASH_TABLE		
SMFR_POINTER		
SMFR_COUNT		

Constants

AST_CHANGE
DIR_STATUS
DIR_STATUS_MASK
DIR_STATUS_NOTMASK
DIR_UNINITIALIZED

3.2.29.5 Limitations

3.2.29.6 Listing

```
DATA ACT(DASTE#, OFFSET) RETURNS (RC);
```

```
PROGRAM ACT;
     DECLARE
          WORD (ASTE#, I, NEXT, HASH_VAL);

     /*    ALLOCATE AN ASTE ENTRY                                          */

     /*    HEAD OF FREE ASTE CHAIN IS IN ASTE 0                            */

     IF AST_CHAIN(0) = 0;
       THEN:

          /*    NO FREE ASTE#'S - LOOK ON AGE CHAIN                        */

          I := 0;

          CYCLE
               NEXT := AST_AGE_CHAIN(I);
          .... EXIT WHEN AST_AGE_CHAIN(NEXT) = 0;
               I := NEXT;
          END;

          DEACT(NEXT);
     END;

     /*    A FREE ASTE# EXISTS                                             */

     ASTE# := AST_CHAIN(0);

     /*    REMOVE THIS ASTE FROM THE FREE CHAIN AND AGE                    */

     AST_CHAIN(0) := AST_CHAIN(ASTE#);
     AST_AGE_CHAIN(ASTE#) := AST_AGE_CHAIN(0);
     AST_AGE_CHAIN(0) := ASTE#;

     /*    UPDATE HASH DATA BASE                                           */

     HASH_VAL := PREHASH(DIR_DISK(OFFSFT));
     AST_CHAIN(ASTE#) := HASH_TABLE(HASH_VAL);
     HASH_TABLE(HASH_VAL) := ASTE#;

     /*    CLEAN UP AST ENTRY                                              */

     AST_ADR(ASTE#) := 0;
     AST_DES_COUNT(ASTE#) := 0;
     AST_CPL(ASTE#) := 0;

     /*    FILL IN AST ENTRY                                               */

     AST_CLASS(ASTE#) := DIR_CLASS(OFFSET); /* SETS TYPE, STATUS */
     AST_CAT(ASTE#) := DIR_CAT(OFFSET);
     AST_DISK(ASTE#) := DIR_DISK(OFFSET);
     AST_SIZE(ASTE#) := DIR_SIZE(OFFSET);

     IF (DIR_STATUS(OFFSFT) & DIR_STATUS_MASK) = DIR_UNINITIALIZED;
       THEN: DIR_STATUS(OFFSET) := (DIR_STATUS(OFFSET) & DIR_STATUS NOTMASK);

          /*    DIRECTORY HAS BEEN WRITTEN INTO - MUST SET CHANGE BIT      */

          AST_CHANGE(DASTE#) := (AST_CHANGE(DASTE#) | AST_CHANGED);
      END;

      /*    INITIALIZE SEMAPHORE ASSOCIATED WITH SEGMENT                   */

      SMFR_COUNT(ASTE#) := 1;
      SMFR_POINTER(ASTE#) := 0;
      RC := ASTE#;
```

3.2.30 Deactivate Segment (DEACT)

The Deactivate Segment CPC, DEACT, is a kernel level internal SKCPP function that is only called by other kernel level internal functions. DEACT calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.30.1 Description

DEACT removes a specified segment from the age chain and frees its active segment table entry. First, it runs through the age chain to find entry ASTE#. Starting with INDEX equal to zero, it repeatedly resets NEXT to AST_AGE_CHAIN(INDEX) and INDEX to NEXT until NEXT equals ASTE#. It links INDEX, the aste just before the specified ASTE to the entry after the specified ASTE by setting AST_AGE_CHAIN(INDEX) to AST_AGE_CHAIN(ASTE#). Setting AST_AGE_CHAIN(ASTE#) to zero marks it unaged.

Next, since an uninitialized segment cannot be deactivated, if AST_STATUS(ASTE#) logical and AST_STATUS_MASK equals AST_UNITIALIZED, DEACT calls SWAPIN. SWAPIN places the segment in main memory and also causes it to be initialized. Then, if AST_ASR(ASTE#), which holds the segment's main memory address, is non-zero, SWAPOUT is called to remove the segment from main memory.

DEACT then removes the ASTE from the hash data base. It calls PREHASH which finds the HASH_VALue associated with the segment's disk address AST_DISK(ASTE#). If HASH_TABLE(HASH_VAL), the head of the chain of aste's with hash values of HASH_VAL, equals ASTE#, ASTE can be removed from the hash data base simply by resetting HASH_TABLE(HASH_VAL) to AST_CHAIN(ASTE#). Otherwise, DEACT must run through the chain until it finds ASTE. It starts by setting HASH_VAL to AST_CHAIN(ASTE#) and repeatedly setting NEXT to AST_CHAIN(HASH_VAL) and HASH_VAL to NEXT until NEXT equals ASTE#.

Now, ASTE can be removed from the HASH chain by letting AST_CHAIN(HASH_VAL) equal AST_CHAIN(ASTE#).

Finally, DEACT places ASTE# at the head of the free chain. It links it to the first aste on the chain by setting AST_CHAIN(ASTE#) to AST_CHAIN(Ø) and resetting the head of the chain, AST_CHAIN(Ø) to ASTE#. DEACT is then exited.

```
Function:  DEACTIVATE
Parameters:  DEACTIVATE(aste#)
Effect:
UNAGE('AST_AGE_CHAIN'(0), aste#);
IF  'AST_STATUS'(aste#) = UNINITIALIZED;
THEN:  SWAPIN(aste#);
   SWAPOUT(aste#);
ELSE:
   IF  'AST_ADR'(ASTE#) ≠ 0;
   THEN:  SWAPOUT(aste#);
   END;
END;
HASH(AST_DISK(aste#)) = 0;
AST_CHAIN(aste#) = 'AST_CHAIN'(0);
AST_CHAIN(0) = aste#;

Function:  AGE
Parameters:  AGE(aste#)
Effect:
AST_AGE_CHAIN(aste#, = 'AST_AGE_CHAIN'(0);
AST_AGE_CHAIN(0) = aste#;
AST_AGE(aste#) = AGED;


Function:  UNAGE
Parameters:  UNAGE(vaste#) = aste#)
Effect:
IF  'AST_AGE_CHAIN'(vaste#) = aste#;
THEN:  AST_AGE_CHAIN(vaste#) = AST_AGE_CHAIN(aste#);
   AST_AGE(aste#) = UNAGED;
ELSE:  UNAGE('AST_AGE_CHAIN'(vaste#, aste#);
END;

Function:  NEXTASTE
Parameters:  NEXTASTE(aste#)
Effect:
IF  AST_AGE_CHAIN(aste#) = 0;
THEN:  aste#;
ELSE:  NEXTASTE(AST_AGE_CHAIN(aste#));
END;
```

3.2.30.2 N/A

3.2.30.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
DELETSEG	SWAPIN
ACT	SWAPOUT
	PREHASH

3.2.30.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function DEACT. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
AST_CHAIN	ASTE#	HASH_VAL
AST_AGE_CHAIN		INDEX
AST_STATUS		NEXT
AST_ADR		
HASH_TABLE		

Constants

AST_STATUS_MASK
AST_UNINITIALIZED

3.2.30.5 Limitations

None

3.2.30.6 Listing

```
DATA DEACT(ASTE#);


PROGRAM DEACT;
     DECLARE
          WORD (HASH_VAL, INDEX, NEXT);

     /*    REMOVE FROM AGE CHAIN                                          */

     INDEX := 0;
```

```
CYCLE
     NEXT := AST_AGE_CHAIN(INDEX);
.... EXIT WHEN NEXT = ASTE#;
     INDEX := NEXT;
END;

AST_AGE_CHAIN(INDEX) := AST_AGE_CHAIN(ASTE#);
AST_AGE_CHAIN(ASTE#) := 0;

/*    CAN NOT DEACTIVATE AN UNINITIALIZED SEGMENT                                 */

IF (AST_STATUS(ASTE#) & AST_STATUS_MASK) = AST_UNINITIALIZED;
  THEN: SWAPIN(ASTE#);
END;

/*    SWAPOUT IF IN MAIN MEMORY                                                   */

IF AST_ADR(ASTE#) ¬= 0;
  THEN: SWAPOUT(ASTE#);
END;

/*    REMOVE THIS ASTE FROM HASH TABLE OR HASH CHAIN                              */

HASH_VAL := PREHASH(AST_DISK(ASTE#));

IF HASH_TABLE(HASH_VAL) = ASTE#;
  THEN: HASH_TABLE(HASH_VAL) := AST_CHAIN(ASTE#);
  ELSE: HASH_VAL := HASH_TABLE(HASH_VAL);

     CYCLE
          NEXT := AST_CHAIN(HASH_VAL);
     .... EXIT WHEN NEXT = ASTE#;
          HASH_VAL := NEXT;
     END;

     AST_CHAIN(HASH_VAL) := AST_CHAIN(ASTE#);
END;

/*    ADD THIS AST ENTRY TO THE FREE CHAIN                                        */

AST_CHAIN(ASTE#) := AST_CHAIN(0);
AST_CHAIN(0) := ASTE#;
```

3.2.31 Swap Segment In (SWAPIN)

The Swap Segment in CPC, SWAPIN, is a kernel level internal SKCPP function that is called by both user level external and kernel level internal functions. SWAPIN calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.31.1 Description

SWAPIN swaps a specified active segment into main memory. First, it allocates a free block of space for the segment. It assigns BLOCK# the value of MBT_CHAIN(∅) logical and MBT_CHAIN_MASK (the flags, chain, and aste entries all share a word on the MBT). If BLOCK# equals END_BLOCK#, there are no more blocks on the free chain, so a segment must be swapped out. SWAPIN finds the segment which has been eligible for swapping out longest by running through to

the end of the swap chain. It sets I equal to zero. Then, it repeatedly resets NEXT to the next element in the chain, AST_SWAP_CHAIN(I) and I to NEXT until AST_SWAP_CHAIN(NEXT) is zero. Having found the end of the swap chain, it corrects BLOCK# to AST_ADR(NEXT), the main memory address of the segment it will get rid of, and calls SWAPOUT to remove NEXT from main memory. There is now at least one block on the free chain. Assigning MBT_CHAIN(Ø) the value of MBT_CHAIN(ASTE#) removes BLOCK# from the head of the chain and setting MBT_FLAGE(BLOCK#) to ALLOCATED marks the block allocated.

If the segment is uninitialized, SWAPIN initializes it rather than performing disk I/O. If AST_STATUS(ASTE#) logical and AST_STATUS_MASK (status shares a byte with four other AST entries) equal AST_UNINITLIZED, INITSEG is called to reset the segment to all zeros and remove all ACL elements if the segment is a directory. Otherwise, DISKIO is called to read the segment from the disk. A P is performed in the DISK_SMFR which stalls the process until the disk operation is complete and a V is performed on the disk semaphore.

SWAPIN then updates the data base. AST_ADR(ASTE#) is assigned BLOCK#, the main memory address of the segment, and MBT_ASTE(BLOCK#) gets ASTE# logical or ALLOCATED. Then, since no descriptors for the segment exist yet, SWAPIN marks the segment eligible for swapping out. It adds it to the head of the swap chain by letting AST_SWAP_CHAIN(ASTE#) equal AST_SWAP_CHAIN(Ø) and by letting AST_SWAP_CHAIN(Ø) equal ASTE#. Resetting AST_UNLOCK(ASTE#) to AST_UNLOCK(ASTE#) logical or AST_UNLOCK_FLAG shows that no descriptors exist for the segment and completes the operation.

```
Function:  SWAPIN
Parameters:  SWAPIN(aste#)
Effect:
Let size = AST_SIZE(aste#);
Let block# = FINDFREE('MBT_CHAIN'(0), size);
ALLOCBLOCK('MBT_CHAIN'(0), block#);
IF  'AST_STATUS'(aste#) = UNINITIALIZED;
THEN:  INITSEG(aste#, block#);
   AST_STATUS(aste#) = INITIALIZED:
   AST_CHANGE(block#) = CHANGED;
ELSE:  DISKIO(aste#, block#, DISK_READ);
   AST_CHANGE(block#) = UNCHANGED;
   P(DISK_SEMAPHORE);
END;
AST_ADR(aste#) = block#;
MBT_ASTE(block#) = aste#;
```

```
UNLOCK(aste#);

Function:  FINDFREE
Parameters:  FINDFREE(block#, size)
value:
IF  block# = END_BLOCK#;
THEN:  RESTART;
ELSE:
   IF  MBT_SIZE(block#) = size;
   THEN:  block#;
   ELSE:  FINDFREE(MBT_CHAIN(block#, size);
   END;
END;

Function:  ALLOCMEM
Parameters:  ALLOCMEM(vblock#, block#)
Effect:
IF  'MBT_CHAIN'(vblock#) = block#;
THEN:  MBT_CHAIN(vblock#) = MBT_CHAIN(block#);
   MBT_FLAGE(block#) = ALLOCATED;
ELSE:  ALLOCMEM('MBT_CHAIN'(vblock#), block#);
END;
```

3.2.31.2 N/A

3.2.31.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
DEACT	INITSEG
GETDIR	SWAPOUT
READIR	P
WRITEDIR	DISKIO
DSEARCH	
ENABLE	

3.2.31.4 Data Base Organization

Listed below are Security Kernel data base references and constants used by the function SWAPIN. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
AST_SWAP_CHAIN	ASTE#	BLOCK#
AST_ADR		NEXT
AST_STATUS		I
AST_UNLOCK		
MBT_CHAIN		
MBT_FLAG		
MBT_ASTE		

Constants

ALLOCATED
AST_STATUS_MASK
AST_UNINITIALIZED
AST_UNLOCK_FLAG
DISK_READ
DISK_SMFR
END_BLOCK#
MBT_CHAIN_MASK

3.2.31.5 Limitations

None

3.2.31.6 Listing

```
DATA SWAPIN(ASTE#);
     DECLARE
          PROCEDURE ACCEPTS (WORD, WORD) (INITSEG);


PROGRAM SWAPIN;
     DECLARE
          WORD (BLOCK#, I, NEXT);

     /*    LOOK FOR A FREE BLOCK                                              */

     BLOCK# := (MBT_CHAIN(0) & MBT_CHAIN_MASK);

     IF BLOCK# = END_BLOCK#;

          /*    NO FREE BLOCKS - LOOK AT SWAP CHAIN FOR SOMETHING TO SWAPOUT  */

       THEN: I := 0;

          CYCLE
               NEXT := AST_SWAP_CHAIN(I);
          .... EXIT WHEN AST_SWAP_CHAIN(NEXT) = 0;
               I := NEXT;
          END;
```

```
        BLOCK# := AST_ADR(NEXT);
        SWAPOUT(NEXT);
    END;

    /*    REMOVE BLOCK FROM FREE CHAIN                                                    */

    MBT_CHAIN(0) := MBT_CHAIN(BLOCK#);
    MBT_FLAGS(BLOCK#) := ALLOCATED;

    /*    IF SEGMENT IS UNINITIALIZED DO NOT PERFORM DISK I/O                             */

    IF (AST_STATUS(ASTE#) & AST_STATUS_MASK) = AST_UNINITIALIZED;
      THEN: INITSEG(ASTE#, BLOCK#);
      ELSE:

        /*    START DISK I/O AND WAIT FOR COMPLETEION                                     */

        DISKIO(AST_DISK(ASTE#), BLOCK#, AST_SIZE(ASTE#), DISK_READ);
        P(DISK_SMFR);
    END;

    /*    UPDATE AST                                                                      */

    AST_ADR(ASTE#) := BLOCK#;
    MBT_ASTE(BLOCK#) := (ASTE# | ALLOCATED);

    /*    PUT SEGMENT ON SWAP CHAIN                                                       */

    AST_SWAP_CHAIN(ASTE#) := AST_SWAP_CHAIN(0);
    AST_SWAP_CHAIN(0) := ASTE#;
    AST_UNLOCK(ASTE#) := (AST_UNLOCK(ASTE#) | AST_UNLOCK_FLAG);
```

3.2.32 Swap Segment Out (SWAPOUT)

The Swap Segment Out CPC, SWAPOUT, is a kernel level internal SKCPP function that is only called by other kernel level internal functions. SWAPOUT calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.32.1 Description

SWAPOUT removes a specified segment from main memory. It sets BLOCK# to AST_ADR(ASTE#), which holds the main memory address of the segment, and AST_ADR_(ASTE#) to zero to show that the segment is swapped out.

Then, SWAPOUT marks the segment ineligible for swapping out. It goes through the swap chain to find the segment identified by ASTE#. Starting with I equal to Ø and resetting NEXT to AST_SWAP_CHAIN(INDEX) and INDEX to NEXT until NEXT equals ASTE#. INDEX now holds the aste# of the segment before the one designated to be swapped out. SWAPOUT removes ASTE# from the swap chain by setting AST_SWAP_CHAIN(INDEX) to AST_SWAP_CHAIN(ASTE#) and AST_SWAP_CHAIN(ASTE#) to zero. It then resets the unlocked bit to locked by letting AST_UNLOCK(ASTE#) equal AST_UNLOCK(ASTE#) logical and AST_LOCK_MASK (all 1's except for a Ø in the UNLOCK bit).

If the segment has been changed while in main memory, it should be copied onto disk; otherwise SWAPOUT doesn't bother. If AST_CHANGE(ASTE#) logical and AST_CHANGE_MASK equals AST_CHANGED, DISKIO is called to initiate a disk write operation. While the operation is performed, SWAPOUT resets the change bit to AST_CHANGE (ASTE#) logical and AST_UNCHANGED_MASK. To avoid getting ahead of the disk, it performs a P on the DISK_SMFR, stalling the process until the disk write is completed.

SWAPOUT then frees the memory allocated to the segment, putting the block on the free chain in ascending order of block#'s. Starting with INDEX equal to Ø, it runs through the chain until the next block#, MBT_CHAIN(INDEX) logical and MBT_CHAIN_MASK, is greater than BLOCK#, repeatedly resetting INDEX to MBT_CHAIN(INDEX) and MBT_CHAIN_MASK. INDEX now contains the block after which BLOCK# should be inserted. Setting MBT_CHAIN(BLOCK#) equal to MBT_CHAIN(INDEX) and MBT_CHAIN (INDEX) to BLOCK# logical or FREE_MEM places BLOCK# in the proper position on the free chain. SWAPOUT is then exited.

```
Function:  SWAPOUT
Parameters:  SWAPOUT(aste#)
Effect:
Let block# = 'AST_ADR' (aste#);
LOCK(aste#);
AST_ADR(aste#) = 0;
IF  AST_CHANGE(block#) = CHANGED;
THEN:  DISKIO(aste#, block#, DISK_WRITE);
   P(DISK_SEMAPHORE);
END;
FREEMEM('MBT_CHAIN'(0), block#);

Function:  FREEMEM
Parameters:  FREEMEM(vblock#, block#)
Effect:
IF  'MBT_CHAIN(block#) > block#;
THEN:  MBT_CHAIN(block#) = 'MBT_CHAIN'(vblock#);
   MBT_CHAIN(vblock#) = block#;
   MBT_FLAGS(block#) = FREE;
   MBT_ASTE(block#) = 0;
ELSE:  FREEMEM('MBT_CHAIN'(vblock#), block#);
END;
```

3.2.32.2 N/A

3.2.32.3 <u>Interfaces</u>

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
DEACT	P
SWAPIN	DISKIO

3.2.32.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function SWAPOUT. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2

Data Base References

Global References	Function Parameters	Local References
AST_ADR	ASTE#	BLOCK#
AST_SWAP_CHAIN		INDEX
AST_UNLOCK		NEXT
AST_CHANGE		
MBT_CHAIN		

Constants

ALLOCATED
AST_CHANGE
AST_CHANGE_MASK
AST_LOCK_MASK
AST_UNCHANGED_MASK
DISK_SMFR
DISK_WRITE
FREE_MEM
MBT_CHANGE_MASK

3.2.32.5 Limitations

None

3.2.32.6 Listing

```
DATA SWAPOUT(ASTE#):
```

```
PROGRAM SWAPOUT;                                                        |16
     DECLARE                                                            |16
          WORD (BLOCK#, INDEX, NEXT);                                   |16
     BLOCK# := AST_ADR(ASTE#);                                          |40
     AST_ADR(ASTE#) := 0;                                               |64
                                                                        |
     /*    REMOVE FROM SWAP CHAIN                                     */|64
                                                                        |
     INDEX := 0;                                                        |72
                                                                        |
     CYCLE                                                              |72
          NEXT := AST_SWAP_CHAIN(INDEX);                                |114
     .... EXIT WHEN NEXT = ASTE#;                                       |136
          INDEX := NEXT;                                                |144
     END;                                                               |146
                                                                        |
     AST_SWAP_CHAIN(INDEX) := AST_SWAP_CHAIN(ASTE#);                    |210
     AST_SWAP_CHAIN(ASTE#) := 0;                                        |234
     AST_UNLOCK(ASTE#) := (AST_UNLOCK(ASTE#) & AST_LOCK_MASK);          |304
                                                                        |
     /*    ONLY PERFORM DISK WRITE IF SEGMENT HAS CHANGED             */|304
                                                                        |
     IF (AST_CHANGE(ASTE#) & AST_CHANGE_MASK) = AST_CHANGED;            |346
       THEN: DISKIO(AST_DISK(ASTE#), BLOCK#, AST_SIZE(ASTE#), DISK_WRITE);  |442
                                                                        |
          /*    DO BOOKKEEPING AND WAIT FOR I/O TO COMPLETE           */|442
                                                                        |
          AST_CHANGE(ASTE#) := AST_CHANGE(ASTE#) & AST_UNCHANGED_MASK;  |512
          P(DISK_SMPR);                                                 |542
     END;                                                               |542
                                                                        |
     /*    FREE MEMORY ALLOCATED TO SEGMENT                           */|542
                                                                        |
     INDEX := 0;                                                        |550
                                                                        |
     CYCLE                                                              |550
     .... EXIT WHEN (MBT_CHAIN(INDEX) & MBT_CHAIN_MASK) > BLOCK#;       |616
          INDEX := (MBT_CHAIN(INDEX) & MBT_CHAIN_MASK);                 |652
     END;                                                               |654
                                                                        |
     /*    PLACE BLOCK TO BE FREED IN THE CHAIN                       */|654
                                                                        |
     MBT_CHAIN(BLOCK#) := MBT_CHAIN(INDEX);                             |716
     MBT_CHAIN(INDEX) := (BLOCK# | FREE_MEM);                           |750


                                                                        |0
```

3.2.33 Initialize Segment (INITSEG)

The Initialize Segment CPC, INITSEG, is a kernel level internal SKCPP function that is called by another kernel level internal function. INITSEG calls only one other kernel level internal function. It is written in Project SUE System Language.

3.2.33.1 Description

INITSEG initializes a directory or data segment. It sets

AST_ADR(ASTE#), the segment's main memory address, to the BLOCK# supplied as a parameter. It then calls LSD to load the descriptors of the segment identified by ASTE# in the kernel register at DIR_KSR_ADR. This provides access to the segment.

INITSEG then performs the actual initialization. As I goes from Ø to 511, it sets ZERO_ARRAY(I) to Ø. Thus, assuming all segments are SIZE2, 1K bytes, if the segment is a directory, all entires are marked free, and if it is a data segment, it is set to all zeros. Next, INITSEG tests if AST_TYPE(ASTE#) logical and AST_TYPE_MASK (type shares a byte with four other attributes) equal AST_TYPE_DIRECTORY. If so, the segment is a directory and it must place all ACL elements on the free chain. Letting ACL_CHAIN(I) equal I + 1 as I goes from Ø to ACL_MAX links all elements from ACL_MIN = 1 to ACL_MAX to the free chain head, ACL_CHAIN(Ø). INITSEG then sets ACL_CHAIN(ACL_MAX) to Ø to mark the end of the free chain.

To complete the operation, INITSEG updates the AST. AST_STATUS is set to AST_STATUS_NOTMASK. This resets the status bit to Ø, meaning initialized. The change bit is set by assigning to ACT_CHANGE the value of AST_CHANGE logical or AST_CHANGED, so that the segment will be copied to the disk when it is swapped out. INITSEG then returns.

```
Function:  INITSEG
Parameters:  INITSEG(aste#, block#)
Effect:
IF  AST_TYPE(aste#) = DIRECTORY;
THEN:
   IF  (OFFSET_MIN  <= i  <= OFFSET_MAX;
   THEN:  DIR_SIZE(i) = 0;
   END;
      IF  (ACL#_MIN <= j <= ACLE#_MAX;
      THEN:  ACL_CHAIN(j) = (j+1) MODULO ACLE#_MAX;
      END;
ELSE:  segment_contents = 0;
END;
```

3.2.33.2 N/A

3.2.33.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
SWAPIN	LSD

3.2.33.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function INITSEG. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
AST_ADR	ASTE#	I
AST_TYPE	BLOCK#	
AST_STATUS		
AST_CHANGE		
ACL_CHAIN		

Constants

ACL_MAX
AST_CHANGE
AST_STATUS_NOTMASK
AST_TYPE_DIRECTORY
AST_TYPE_MASK
DIR_KSR_ADR
SDR_WRITE_ACCESS

3.2.33.5 Limitations

None

3.2.33.6 Listing

```
DATA INITSEG(ASTE#, BLOCK#);


PROGRAM INITSEG;
     DECLARE
          WORD (I);
     AST_ADR(ASTE#) := BLOCK#;

     /*    GAIN ACCESS TO SEGMENT                                                    */
```

```
LSD(ASTE#, DIR_KSR_ADR, SDR_WRITE_ACCESS);

/*    THIS ASSUMES ALL SEGS ARE 1K BYTES - WILL CHANGE LATER                */

DO I := 0 TO 511;
    ZERO_ARRAY(I) := 0;
END;

IF (AST_TYPE(ASTE#) & AST_TYPE_MASK) = AST_TYPE_DIRECTORY;
  THEN:

    /*    PUT ALL ACL ELEMENTS ON FREE CHAIN                                */

    DO I := 0 TO ACL_MAX;
        ACL_CHAIN(I) := I + 1;
    END;

    ACL_CHAIN(ACL_MAX) := 0;
END;

/*    UPDATE AST                                                            */

AST_STATUS(ASTE#) := (AST_STATUS(ASTE#) & AST_STATUS_NOTMASK);
AST_CHANGE(ASTE#) := (AST_CHANGE(ASTE#) | AST_CHANGED);
```

3.2.34 Prehash (PREHASH)

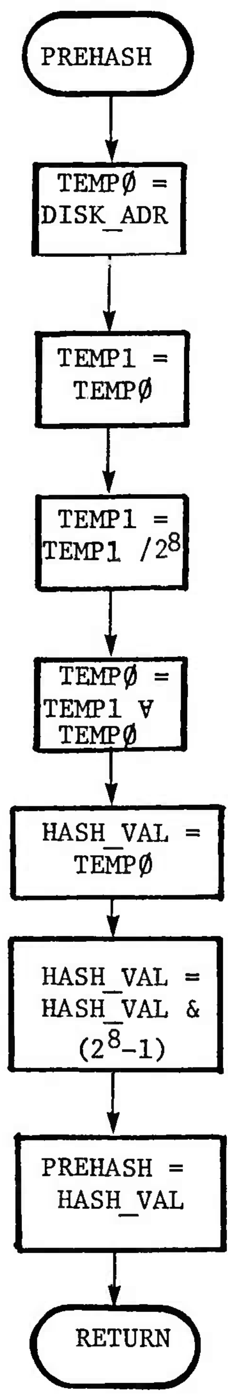
The Prehash CPC, PREHASH, is a kernel level internal SKCPP function that is directly called by other kernel level internal functions. It is written in Project SUE System Language, including the use of the Inline feature.

3.2.34.1 Description

PREHASH converts the disk address of a segment input to it into an index into HASH_TABLE. It hash codes the 16-bit disk address into an 8-bit index equal to the exclusive or of the high order 8-bits with the low order 8-bits of the address.

Function: PREHASH
Possible values: HASH_VAL or 0
Initial value: 0
Parameters: PREHASH(disk_address)

3.2.34.2 Flow Chart



3.2.34.3 Interfaces

Called By	Calls
ACT DEACT	None

3.2.34.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function PREHASH. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
None	DISK_ADR	HASH_VAL

Constants

ASHR1
MOV
SEG_FLAG
XOR10

3.2.34.5 Limitations

None

3.2.34.6 Listing

```
DATA PREHASH(DISK_ADR) RETURNS (HASH_VAL);



 PROGRAM PREHASH;
      INLINE(MOV, DISK_ADR, 0, 0);
      INLINE(MOV, 0, 0, 0, 1);
      INLINE(ASHR1);
      INLINE("FFF8");
      INLINE(XOR10);
      INLINE(MOV, 0, 0, HASH_VAL);
      HASH_VAL := (HASH_VAL & "00FF");
```

3.2.35 Hash (HASH)

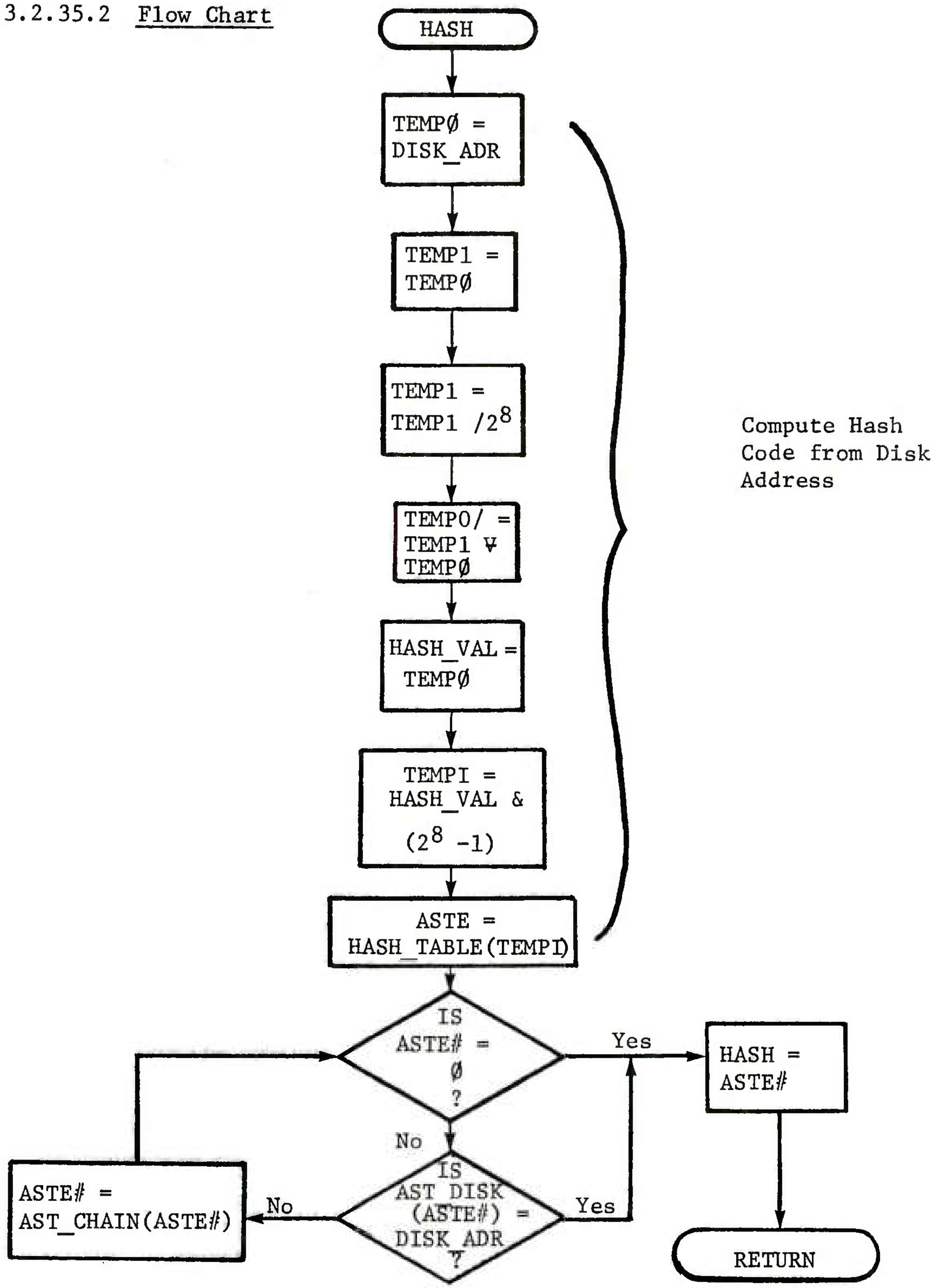
The Hash CPC, HASH, is a kernel level internal SKCPP function that is directly called by both user level and other kernel level internal functions. It is written in Project SUE System Language, including the use of the Inline feature.

3.2.35.1 Description

HASH converts the disk address of a segment input to it into the aste# of the segment, if the segment is active. It hash codes the 16-bit disk address into an 8-bit index into HASH_TABLE equal to the exclusive or of the high order 8-bits with the low order 8-bits of the address. This index is then used to retrieve the associated aste#. If the disk address stored in the AST for this aste# matches the input disk address, this is the required aste#. If it does not match, any chains caused by hashing collisions must be run. This process continues until a match occurs or an aste# of zero is retrieved, indicating that the segment is not active.

Function: HASH
Possible values: aste# or 0
Initial value: 0
Parameters: HASH(disk_address)

3.2.35.2 Flow Chart



3.2.35.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
DELETE	None
STARTP	
CHANGEO	
DELETSEG	
SOADD	
CONNECT	

3.2.35.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function HASH. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
HASH_TABLE	DISK_ADR	HASH_VAL
AST_DISK		ASTE#
AST_CHAIN		

Constants

ASHR1
MOV
XOR10

3.2.35.5 Limitations

None

3.2.35.6 Listing

```
DATA HASH(DISK_ADR) RETURNS (ASTE#);


PROGRAM HASH;
     DECLARE
          WORD (HASH_VAL);

     /*     HASH DISK_ADR                                                  */
```

```
    INLINE(MOV, DISK_ADR, 0, 0);
    INLINE(MOV, 0, 0, 0, 1);
    INLINE(ASHR1);
    INLINE("FFF8");
    INLINE(XOR10);
    INLINE(MOV, 0, 0, HASH_VAL);
    ASTE# := HASH_TABLE(HASH_VAL & "00FF");

    CYCLE
....        RETURN WHEN ASTE# = 0;
....        RETURN WHEN AST_DISK(ASTE#) = DISK_ADR;
            ASTE# := AST_CHAIN(ASTE#);
    END;
```

3.2.36 Load Segment Descriptors (LSD)

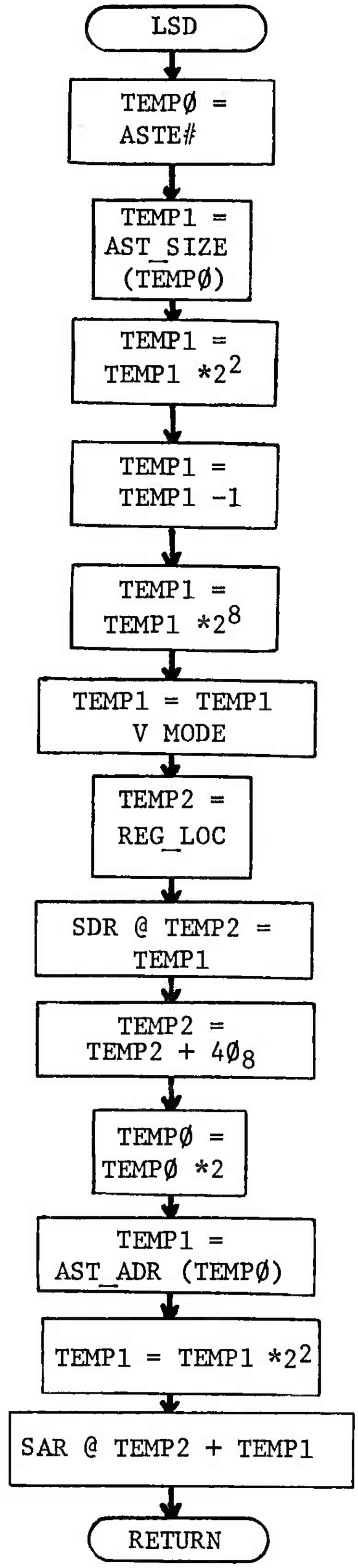
The Load Segment Descriptors CPC, LSD, is a kernel internal SKCPP function that is called by both user level external functions and kernel level internal functions. It is written in PAL-11 assembly language.

3.2.36.1 Description

LSD constructs segmentation descriptors for a specified segment and loads them in a specified register. It converts the AST_SIZE into a form useable in the segment length filed. Since the size is in blocks which omit the eight low-order bits of the length, and the descriptors omit only six low-order bits, the size is multiplied by 2^2. It is decremented because of the MMU's excess one's notation. The size word is then rotated so that the 8 low-order bits become the high-order bits. The register at REG_LOC is assigned the inclusive or of the size word and MODE; thus, its high order byte is the segment length field and its low order byte contains written-into and access control information. REG_LOC, the pointer to the descriptor register, is increased by 40_8 to point to the address register. ASTE# is multiplied by 2 before being used as a pointer to the AST_ADR array of two-byte words. The address is converted from memory blocks to a descriptor address with six low-order bits omitted and assigned to the segment address register. This completes the operation.

Function: LSD
Parameters: LSD(process#, block#, reg#, mode)
Effect:
PS_SAR (process#, reg#) = block#;
PS_SDR (process#, reg#) = MIB_SIZE(block#), mode;

3.2.36.2 <u>Flow Chart</u>



3.2.36.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
ENABLE	None
GETDIR	
READIR	
WRITEDIR	
SOADD	
DSEARCH	
INITSEG	
STARTP	
RUN	

3.2.36.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function LSD. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
AST_SIZE	ASTE#	ASTSIZE
AST_ADR	REG_LOC	ASTDR
	MODE	

Constants

ADD
ASL
DEC
JMP
MOV
MOVB
SWAB

3.2.36.5 Limitations

None

3.2.36.6 Listing

```
                                                          PAGE    1

STMT  SOURCE STATEMENT

      R0=%0
      R1=%1
      R2=%2
      R3=%3
      R4=%4
      R5=%5
      R6=%6
      PC=%7
      ;
      ;          LSD: LOAD_SEGMENT_DESCRIPTOR(ASTE#, REG_LOC, MODE);
      ;
      ASTADR=007400                              ;BASE OF AST_ADR ARRAY
      ASTSIZ=005000                              ;BASE OF AST_SIZE ARRAY
      ;
      LSD:       MOV     R6,R5                ;SUE
                 ADD     #10,R5               ;ENTRY
                 MOV     PC,(R5)+             ;SEQUENCE
                 MOV     -4(R5),R0            ;ASTE#
                 MOVB    ASTSIZ(R0),R1        ;SIZE OF THE SEGMENT
                 ASL     R1                   ;MY BLOCKS TO THEIR'S
                 ASL     R1
                 DEC     R1                   ;MMU USES EXCESS ONES NOTATION
                 SWAB    R1                   ;SLF IS IN LEFT BYTE OF DESCRIPTOR REG
                 BIS     -10(R5),R1           ;OR IN A, W, ED, AND ACF BITS
                 MOV     -6(R5),R2            ;POINTER TO DESCRIPTOR REG
                 MOV     R1,(R2)              ;STORE DESCRIPTOR
                 ADD     #40,R2               ;POINT TO ADDRESS REGISTER
                 ASL     R0                   ;ARRAY OF WORDS
                 MOV     ASTADR(R0),R1        ;BASE ADDRESS OF SEGMENT
                 ASL     R1                   ;MY BLOCKS TO THEIR'S
                 ASL     R1
                 MOV     R1,(R2)              ;STORE IN SEGMENTATION REGISTER
                 JMP     @-12(R5)             ;DONE
                 .END    LSD
```

3.2.37 Disk I/O (DISKIO)

The Disk I/O CPC, DISKIO, is a kernel level internal SKCPP function that is called by other kernel level internal functions. It is written in PAL-11 assembly language.

3.2.37.1 Description

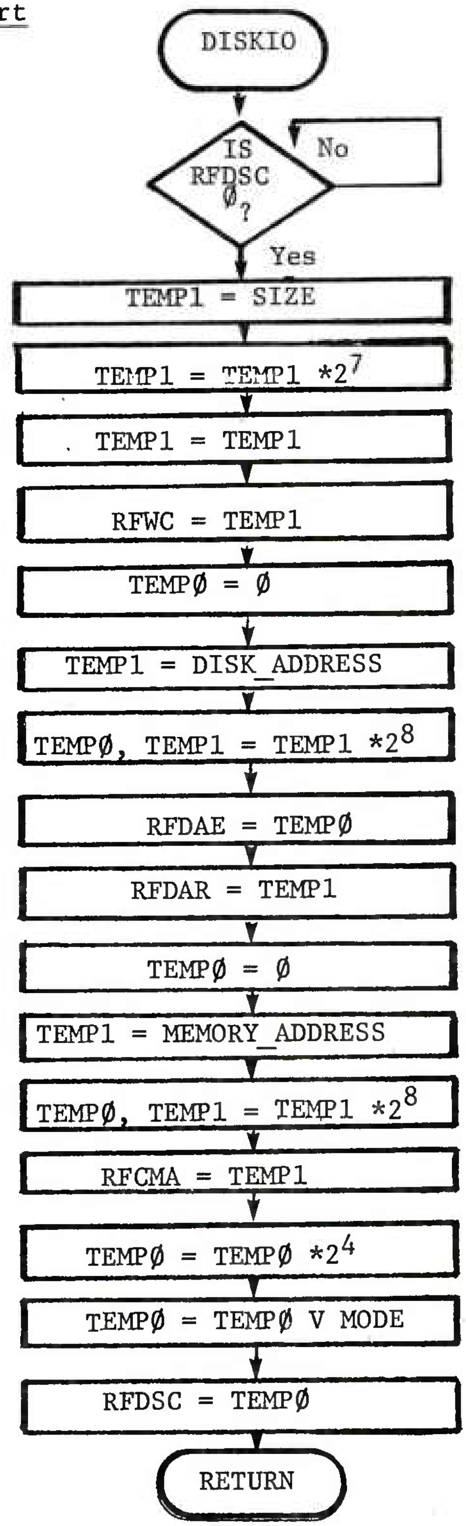
DISKIO initiates a disk input/output operation. If RFDSC, the disk control status, indicates the disk is not ready then there is a hardware failure, which is beyond the scope of the Security Kernel.

Otherwise, it converts the parameters into forms useable by the disk. Multiplying the SIZE in 256-byte blocks by 128 converts the units to words and negating the result yields its two's complement which the disk requires. This value is used as RFDWC, the disk word count. DISKIO then restores the eight low order Ø's to the

DISK_ADDRESS by multiplying it by 2^8. It lets RFDAR, the disk address register, and RFDAE, the disk address extension error register, equal this result. The current memory address, RFCMA, is found by restoring the eight low-order bits to MEMORY_ADDRESS. The disk control status word is then constructed by computing the inclusive or of any extended bits shifted four digits to the left with MODE. Having thus started the disk, DISKIO returns.

```
Function:  DISKIO
Parameters: DISKIO(aste#, block#, command)
Effect:
DISK_ADR = AST_DISK(aste#);
DISK_COUNT = AST_SIZE(aste#)
MEM_ADR = block#;
DISK_COMMAND = command, ENABLE_INTERRUPTS;
```

3.2.37.2 Flow Chart



3.2.37.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
SWAPIN SWAPOUT	None

3.2.37.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function DISKIO. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
None	DISK_ADDRESS	RFDSW
	MEMORY_ADDRESS	RFWC
	SIZE	RFCMA
	MODE	RFDAR
		RFDAE

Constants

ADD
CLR
JMP
MOV
NEG

3.2.37.5 Limitations

The disk control status must be negative for DISKIO to operate.

3.2.37.6 Listing


```
STMT   SOURCE STATEMENT

R0=%0
R1=%1
R2=%2
R3=%3
R4=%4
R5=%5
R6=%6
PC=%7
;
;          DISKIO(DISK_ADDRESS, MEMORY_ADDRESS, SIZE, MODE);
;
RFDSC=177460                         ;DISK CONTROL STATUS
RFWC=177462                          ;WORD COUNT
RFCMA=177464                         ;CURRENT MEMORY ADDRESS
RFDAR=177466                         ;DISK ADDRESS REGISTER
RFDAE=177470                         ;DISK ADDRESS EXTENSION ERROR REGISTER
;
DISKIO:   MOV    R6,R5               ;SUE
          ADD    #12,R5              ;ENTRY
          MOV    PC,(R5)+            ;SEQUENCE
          TSTB   @#RFDSC             ;DISK MUST BE READY
ERROR:    BGE    ERROR               ;OTHERWISE INFINITE LOOP
          MOV    -10(R5),R1          ;SIZE OF SEGMENT IN 256 BYTE BLOCKS
          .WORD  072127 ;ASH R1,#7   ;SLL R1,7
          .WORD  000007              ;SIZE OF SEGMENT IN WORDS
          NEG    R1                  ;DISK REQUIRES 2'S COMPLEMENT OF COUNT
          MOV    R1,@#RFWC           ;MOVE COUNT INTO WORD COUNT REGISTER
          CLR    R0                  ;SET UP FOR SPLITTING DISK ADDRESS INTO 2
          MOV    -4(R5),R1           ;DISK ADDRESS WITH 8 LOW ORDER 0'S REMOVED
          .WORD  073027 ;ASHC R0,#10;SLDL R0,R1,8
          .WORD  000010              ;ADD IN THE LOW ORDER 0'S
          MOV    R0,@#RFDAE          ;MOVE DISK ADDRESS
          MOV    R1,@#RFDAR          ;TO CONTROL REGISTERS
          CLR    R0                  ;SET UP FOR SPLITTING MEM ADDRESS INTO 2
          MOV    -6(R5),R1           ;MEM ADDRESS WITH 8 LOW ORDER 0'S REMOVED
          .WORD  073027 ;ASHC R0,#10;SLDL R0,R1,8
          .WORD  000010              ;ADD IN THE LOW ORDER 0'S
          MOV    R1,@#RFCMA          ;MOVE LOW ORDER WORD INTO CONTROL REGISTER
          .WORD  072027 ;ASH R0,#4   ;SLL R0,4 - MOVE EXTENDED BITS TO RIGHT
          .WORD  000004
          BIS    -12(R5),R0          ;OR IN INT. ENABLE, FUNCTION, AND GO BITS
          MOV    R0,@#RFDSC          ;START DISK
          JMP    @-14(R5)            ;RETURN
          .END   DISKIO
```

3.2.38 Disk Allocation (DALLOC)

The Disk Allocation CPC, DALLOC, is a kernel level internal SKCPP function that is only called by a user level external function. It is written in PAL-11 assembly language.

3.2.38.1 Description

DALLOC allocated space on the disk for a segment being created and sets NEXT_DISK_ADDRESS to the address of the space allocated. Its

parameter is the address of the bit maps table for segments of the desired size; at this time, only size 2, 1K bytes, is implemented. The bit maps table has four entries: the starting and ending addresses of the bit map, the base disk address, and the shift factor, $\log_2$ size.

DALLOC searches the bit map for a zero bit, which indicates a segment not in use. It sets all the bits of TEMP4 to 1. Then, starting at the starting address, it checks if the bit map word equals TEMP4. If not, it had found a word with a zero bit, and can proceed; otherwise, it continues its search. If it reaches the end of the bit map without finding a zero bit, the error is unrecoverable, so it halts.

After finding a word which contains a zero bit, it finds a word number. Since the search operates in auto-increment mode, DALLOC subtracts 2 from TEMP2 to obtain the address of the word with the free bit. TEMP3, the byte number of the word within the bit map, is set to TEMP2 minus the starting address of the bit map.

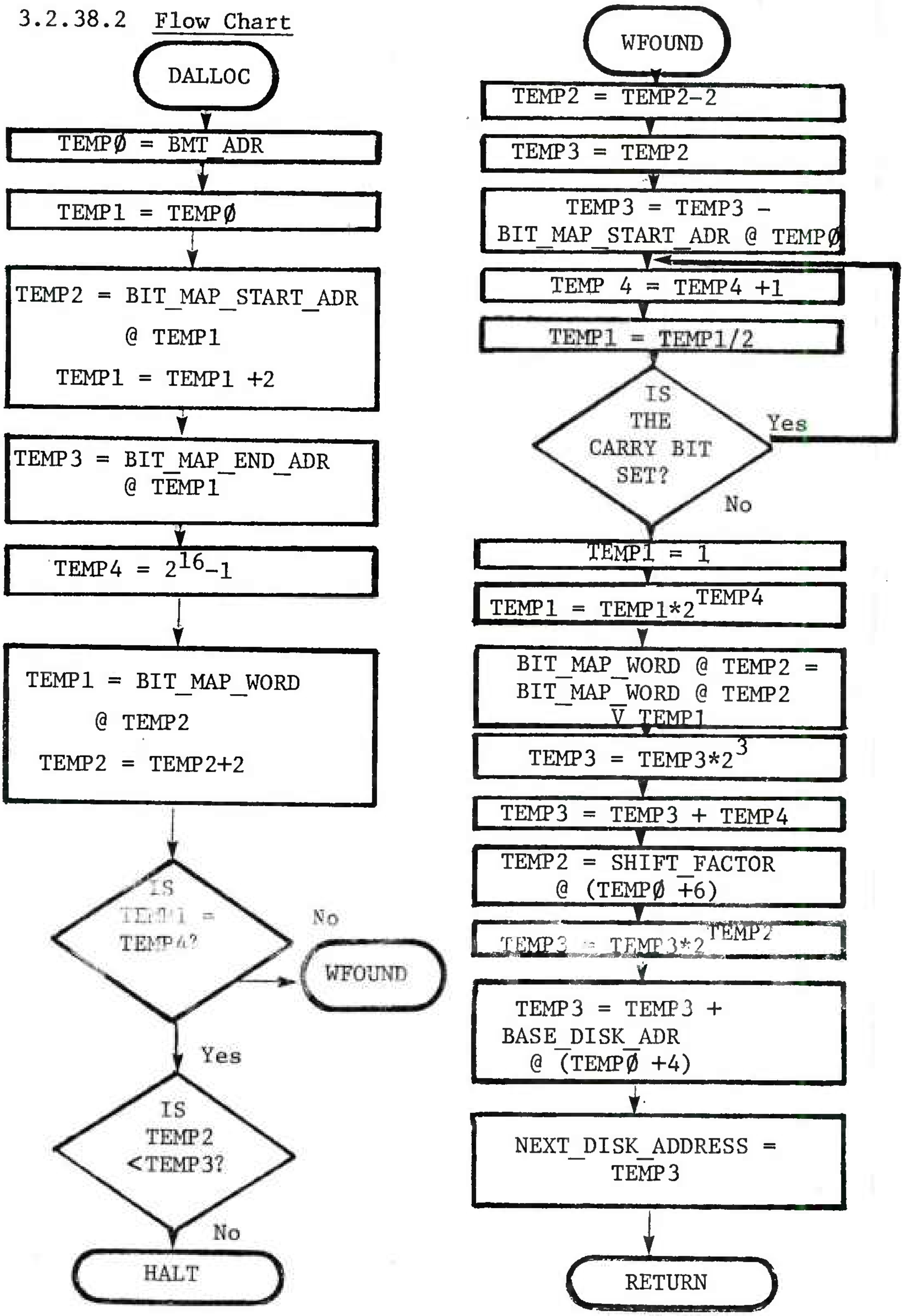
It then searches the word for its first zero bit. It repeatedly performs right shifts, incrementing TEMP4 each time, while the carry bit is set from the rightmost bit before the right shift, TEMP4 now holds the number of the first bit set to zero.

DALLOC now sets the bit for the free segment it found to "in use". It "or's" the word it found with a word containing a 1 in the bit corresponding to the first Ø bit.

DALLOC now translates the byte number and bit number it has found into a disk address. The number of the segment in the disk area has the word number as its high order bits and the bit number as its low order bits. Multiplying this by the size of the segments, that is, 2 SHIFT_FACTOR yields the offset of the allocated disk area from the base of the disk area for segments of this size. Adding BMT(3), the BASE_DISK_ADR to the offset produces a disk address. DALLOC returns this value to the user.

```
Function: DISK_ALLOC
Parameters: DISK_ALLOC(size)
Effect:
(∃k) (('BIT_MAP'(size, k) = 0) &
   (BIT_MAP(size, k) = 1) &
   (NEXT_DISK_ADDRESS = BASE(size) + k*size));
```

3.2.38.2 Flow Chart



3.2.38.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
CREATE	None

3.2.38.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function DALLOC. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
BMT_SIZE	BMT_ADR	DISK_ADR

Constants

ADD
ASR
INC
JMP
MOV
SUB

3.2.38.5 Limitations

None

3.2.38.6 Listing


```
STMT   SOURCE STATEMENT

R0=%0
R1=%1
R2=%2
R3=%3
R4=%4
R5=%5
R6=%6
PC=%7
;
;               DISK_ADR := DALLOC(BMT_ADR);
;
;
DALLOC:         NOP
                MOV     R6,R5                   ;SUE
                ADD     #4,R5                   ;ENTRY
                MOV     PC,(R5)+                ;SEQUENCE
                SUB     #2,R6
                MOV     R4,-(R6)                ;DON'T NEED R4
                MOV     -4(R5),R0               ;GET PARAMETER
                MOV     R0,R1                   ;ADDRESS OF BIT MAP TABLE
                MOV     (R1)+,R2                ;START OF BIT MAP
                MOV     (R1),R3                 ;END OF BIT MAP
                MOV     #177777,R4              ;ALL BITS SET
;
WLOOP:          MOV     (R2)+,R1                ;SEARCH BIT MAP
                CMP     R1,R4                   ;FOR A 0 BIT
                BNE     WFOUND                  ;FOUND ONE
                CMP     R2,R3                   ;END OF MAP?
                BLT     WLOOP                   ;NO
                HALT                            ;UNRECOVERABLE ERROR
;
WFOUND:         SUB     #2,R2                   ;ADDRESS OF WORD WITH FREE BIT
                MOV     R2,R3
                SUB     (R0),R3                 ;ITS WORD NUMBER (RELATIVE TO BASE)
;
BLOOP:          INC     R4                      ;SEARCH
                ASR     R1                      ;FOR FIRST
                BCS     BLOOP                   ;0 BIT
;
;               NOW MUST TURN BIT ON IN MAP
;
                MOV     #1,R1                   ;ONE BIT ON
                .WORD 072104 ;ASH R1,R4         ;SHIFT BIT OVER
                BIS     R1,(R2)                 ;SET USE BIT
;
;               TRANSLATE (WORD#, BIT#, SIZE) INTO A DISK ADDRESS
;
                .WORD 072327 ;ASH R3,3          ;SHIFT WORD NUMBER OVER 3 BITS
                .WORD 3
                ADD     R4,R3                   ;ADD IN BIT NUMBER
                MOV     6(R0),R2                ;SHIFT FACTOR
                .WORD 072302                    ;ASH    R3,R2
```

```
STMT  SOURCE STATEMENT

 53          ADD   4(R0),R3          ;ADD IN BASE ADDRESS OF DISK AREA
 54          MOV   R3,4(R5)          ;RETURN TO USER
 55          MOV   (R6)+,R4          ;RESTORE
 56          JMP   @-6(R5)           ;DONE!!
 57          .END  DALLOC
```

3.2.39 Free Disk (DFREE)

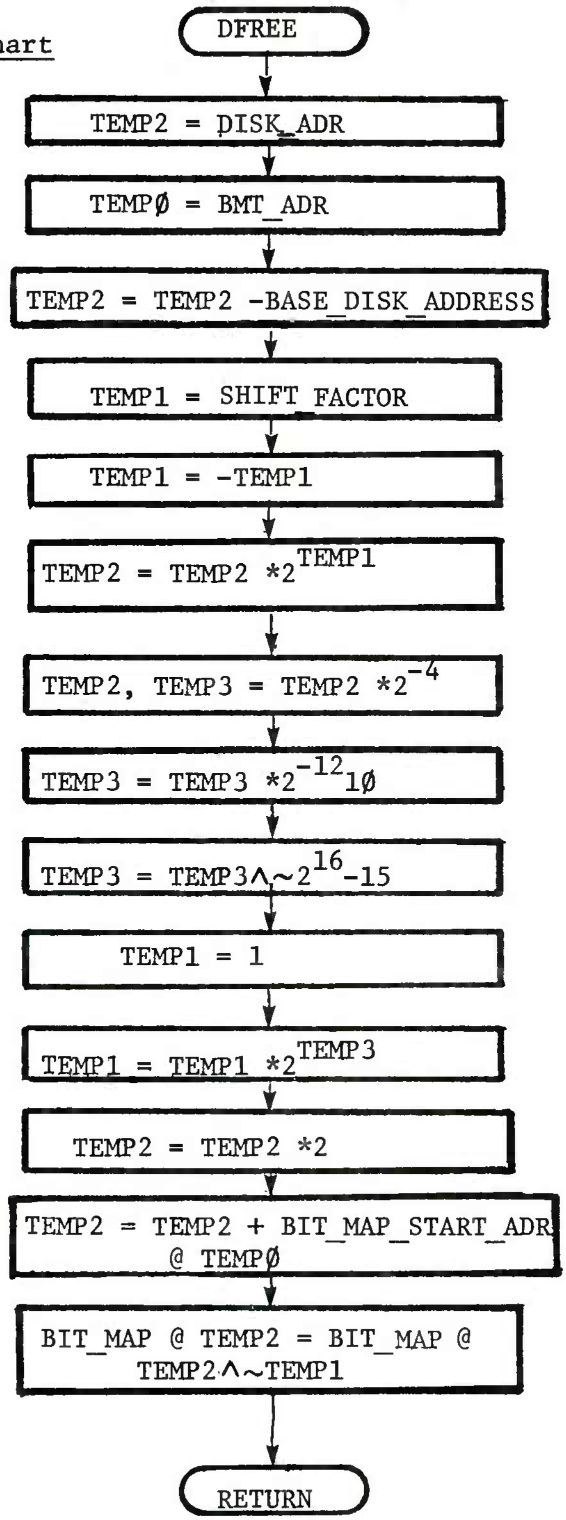
The Free Disk CPC, DFREE, is a kernel level internal SKCPP function that is called by another kernel level internal function. It is written in PAL-11 assembly language.

3.2.39.1 Description

DFREE translates a disk address into a bit address and resets the appropriate bit to mark the disk area free. It subtracts the base disk address in the bit maps table from the disk address supplied to obtain the relative address of the disk area to be freed. Shifting the relative address to the right by the shift factor is the equivalent of dividing by the size: it yields the number of the segment within the disk space. DFREE then performs a combined right shift on this result. This separates the four low-order bits, identifying the bit number, from the high order bits, which identify the word# of the segment's bit in the bit map. The bit# is then moved to the right, and since the left-most bit is duplicated in the right shift, the 12 left bits of the bit# are cleared. DFREE creates a word, TEMP1, with only the bit corresponding to the bit# set to 1. It finds the relative byte address of word# by multiplying word# by 2, and the absolute address, TEMP2 by adding the bit map starting address which it finds in the bit maps table. It then uses TEMP1 to clear the bit at TEMP2 which corresponds to the area to be freed. This updates the bit map and marks the disk space not in use.

```
Function:   DISK_FREE
Parameters:  DISK_FREE(disk_address, size)
Effect:
Let k = (disk_address - BASE(size))/size;
BIT_MAP(size, k) = 0;
```

3.2.39.2 <u>Flow Chart</u>



3.2.39.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
DELETSEG	None

3.2.39.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function DFREE. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
BMT_SIZE	DISK_ADR	
	BMT_ADR	

Constants

ADD
ASL
JMP
MOV
NEG
SUB

3.2.39.5 Limitations

None

3.2.39.6 <u>Listing</u>

```
                                                                    PAGE    1
STMT  SOURCE STATEMENT

      R0=%0
      R1=%1
      R2=%2
      R3=%3
      R4=%4
      R5=%5
      R6=%6
      PC=%7
      ;
      ;         DFREE(DISK_ADR, BMT_ADR);
      ;
      DFREE:    NOP
                MOV    R6,R5                ;SUE
                ADD    #6,R5                ;ENTRY
                MOV    PC, (R5)+            ;SEQUENCE
                SUB    #2,R6
                MOV    -4(R5),R2            ;DISK ADDRESS
                MOV    -6(R5),R0            ;BMT ADDRESS PARAMETER
                SUB    4(R0),R2             ;SUTRACT OUT BASE DISK ADDRESS
                MOV    6(R0),R1             ;GET SHIFT FACTOR
                NEG    R1                   ;NEGATIVE FOR RIGHT SHIFT
                .WORD  072201               ;SRL    R2,R1
                .WORD  073227 ;ASHC R2,-4   ;SRDL R2,3,4
                .WORD  -4                   ;SEPERATE WORD#, BIT#
                .WORD  072327 ;ASH R3,-12   ;SRL R3,12
                .WORD  -14                  ;GET BIT NUMBER OVER TO RIGHT
                BIC    #177760,R3           ;ASH IS ARITHMETIC, NOT LOGICAL, SHIFT
                MOV    #1,R1                ;ONE 1 BIT
                .WORD  072103 ;ASH R1,R3    ;SRL R1,0(R3)
                ASL    R2                   ;WORD# TO BYTE#
                ADD    (R0),R2              ;ADD IN START ADDRESS OF BIT MAP
                BIC    R1,(R2)              ;CLEAR USE BIT
                JMP    @-10(R5)             ;DONE
                .END   DFREE
```

3.2.40 <u>Sleep (SLEEP)</u>

The Sleep CPC, SLEEP, is a kernel level internal SKCPP function that is called by both another kernel level internal function and user level external functions. SLEEP calls a kernel level internal function. It is written in Project SUE System Language, including the Inline feature.

3.2.40.1 <u>Description</u>

SLEEP finds the next proeess which is ready to run, Using Inline code, it sets the priority level low to enable interrupts. Then it loops through the processes, starting with NEXT_PROCESS equal to THE_CURRENT_PROCESS + 1, looking for a ready process. It begins the cycle by checking if NEXT_PROCESS is greater than PROCESS#_MAX. If so, it has reached the last process and goes back to the beginning

by letting NEXT_PROCESS equal PROCESS#_MIN. Next, if PT_ FLAGS (NEXT_PROCESS) logical and PT_FLAGS_MASK equals READY, it has found a ready process and leaves the process finding loop. Otherwise, it increments NEXT_PROCESS and keeps looking.

If NEXT_PROCESS, the ready process it found, is not THE_CURRENT_ PROCESS, SLEEP calls RUN. The segmentation descriptors of the current porcess are saved in the PT and those of NEXT_PROCESS, which becomes the new current process, are loaded. Otherwise, SLEEP simply returns control to the calling program.

```
Function:  SLEEP
Parameters:  SLEEP
Effect:
IF ((∃process#)(PT_FLAGS(process#) = READY);
THEN:
   IF  process# = TCP;
   THEN:  RUN(process#);
   END;
ELSE:  SLEEP
END;
```

3.2.40.2 N/A

3.2.40.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
STOPP	RUN
IPCRCV	
P	

3.2.40.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function SLEEP. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph, 3.3.2.

Data Base References

Global References	Function Parameters	Local References
PT_FLAGS	None	NEXT_PROCESS

Global References	Function Parameters	Local References
THE_CURRENT_PROCESS		

Constants
PROCESS#_MAX
PROCESS#_MIN
PT_FLAGS_MASK
READY
SPILLOW

3.2.40.5 Limitations

None

3.2.40.6 Listing

```
DATA SLEEP;
     DECLARE
          WORD (NEXT_PROCESS);


PROGRAM SLEEP;
     INLINE(SPLLOW);

     /*    FIND NEXT PROCESS THAT CAN RUN - ROUND ROBIN SCHEDULER                  */

     NEXT_PROCESS := THE_CURRENT_PROCESS + 1;

     CYCLE

          IF NEXT_PROCESS > PROCESS#_MAX;
            THEN: NEXT_PROCESS := PROCESS#_MIN;
          END;

          IF (PT_FLAGS(NEXT_PROCESS) & PT_FLAGS_MASK) = READY;
            THEN:
     ....       EXIT;
          END;

          NEXT_PROCESS := NEXT_PROCESS + 1;
     END;

     IF NEXT_PROCESS ¬= THE_CURRENT_PROCESS;
       THEN: RUN(NEXT_PROCESS);
     END;
```

3.2.41. Run (RUN)

The Run CPC, RUN, is a kernel level internal SKCPP function that is called by another kernel level internal function. RUN calls only kernel level internal functions. It is written in Project SUE System Language.

3.2.41.1 Description

RUN saves information about the current process and prepares to run the next process. Using Inline code, it sets the priority level high to block interrupts.

It then looks for changes, looping through all segmentation registers, REG# going from Ø to REG#_MAX. It sets X to the correct number to access the segmentation registers: REG# + REG_CONSTANT if REG# is greater than CROSS_REG# and REG# otherwise. RUN then checks if the change bit is set, if SDR(X) logical and SDR_CHANGE_MASK equals SDR_CHANGED. If so, VAR is assigned the value of SAR(X), the main memory address of the segment. Since the segmentation registers omit the six low order bits of the address (assumed to be Ø) and the memory block table omits the eight low order bits thereof, two arithmetic shift rights are performed, using Inline code, on VAR. Then, if VAR is less than END_BLOCK#, the change bit in the AST entry for the segment is set. VAR gets the aste# from MBT_AST(VAR). AST_CHANGE(VAR) is then reset to AST_CHANGE(VAR) logical or MBT_CHANGED. RUN then continues its loop through the segmentation registers.

```
Function:   RUN
Parameters:   RUN(process#)
Effect:
IF   (reg#)(SDR(reg#)) & SDR_CHANGE_MASK = SDR_CHANGED;
THEN:
   Let VAR = 4*SAR(reg#);
   Let block# = MBT_AST(var);
   AST_CHANGE(block#) = 'AST_CHANGE'(block#)  AST_CHANGED;
END;
PT_KSAR1(TCP) = KSAR1;
PT_KSDR1(TCP) = KSDR1;
LSD(PT_PS_ASTE#(process#) PS_KSR_ASR,WRITE);
```

Next, RUN saves the kernel segmentation register 1. It sets (THE_CURRENT_PROCESS) to KSDR1 and PT_KSAR1(THE_CURRENT_PROCESS) to KSAR1. It then loads the descriptor for the next process' PS at PS_KSAR_ADR with a call to LSD. Finally, RUN calls SWAP, to make the next process the current process.

3.2.41.2 N/A

3.2.41.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
SLEEP	LSD SWAP

3.2.41.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function RUN. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
PT_KSAR1	NEXT_PROCESS	X
PT_KSDR1		
AST_CHANGE		
MBT_ASTE		
SDR		
SAR		
THE_CURRENT_PROCESS		

Constants

ASR	REG#_MAX
AST_CHANGE	SDR_CHANGE_MASK
CROSS_REG#	SDR_CHANGED
END_BLOCK#	SDR_WRITE_ACCESS
PS_KSR_ADR	SPLHIGH
REG_CONSTANT	SPLLOW

3.2.41.5 Limitations

None

3.2.41.6 Listing

```
DATA RUN(NEXT_PROCESS);
     DECLARE
          PROCEDURE ACCEPTS (WORD, WORD) (SWAP),
          WORD (REG#, X, VAR);



PROGRAM RUN;

     /*    BLOCK INTERRUPTS                                                       */

     INLINE(SPLHIGH);

     /*    NO NEED TO SAVE PROCESS REGISTERS BECAUSE A COPY IS IN THE PROCESS SEGMENT,  *
      * BUT CHANGE BIT MUST BE INSPECTED                                          */

     DO REG# := 0 TO REG#_MAX;

          IF REG# > CROSS_REG#;
            THEN: X := REG# + REG_CONSTANT;
            ELSE: X := REG#;
          END;

          IF (SDR(X) & SDR_CHANGE_MASK) = SDR_CHANGED;
            THEN: VAR := SAR(X);
               INLINE(ASR, VAR); /* THEIR BLOCKS TO MINE */
               INLINE(ASR, VAR);

               IF VAR < END_BLOCK#;
                 THEN: VAR := MBT_ASTE(VAR);
                    AST_CHANGE(VAR) := (AST_CHANGE(VAR) | AST_CHANGED);
               END;

          END;

     END;

     /*    SAVE KSR1                                                              */

     PT_KSAR1(THE_CURRENT_PROCESS) := KSAR1;
     PT_KSDR1(THE_CURRENT_PROCESS) := KSDR1;

     /*    LOAD DESCRIPTOR FOR NEXT PROCESS'S PS                                  */

     LSD(PT_PS_ASTE#(NEXT_PROCESS), PS_KSR_ADR, SDR_WRITE_ACCESS);

     /*    CLEANER (AND FASTER) TO DO REAL SWAP IN PAL                            */

     SWAP(THE_CURRENT_PROCESS, NEXT_PROCESS);
     INLINE(SPLLOW);
```

3.2.42 Swap (SWAP)

The Swap CPC, SWAP, is a kernel level internal SKCPP function that is called by one other kernel level internal function. It is written in PAL-11 assembly language.

3.2.42.1 Description

SWAP switches from the current process supplied to the next process, as specified. It multiplies the current process parameter by 2 to access the arrays of 2-byte words. It then saves the contents

of general purpose registers Ø to 6 in the process table.

Next, it enters NEXT_PROCESS at the address of TCP, as the current process. The next process parameter is then doubled to access the arrays of words.

SWAP then loads the segmentation registers from the new current process's process segment. It loops through the supervisor registers, setting SARØ through SAR7 equal to PS_SAR(Ø) through PS_SAR(7). Similarly it loops through the user registers and loads SAR8-15 and SDR8-15.

It then loads kernel segmentation registers KSR1, 2, and 3 from the current process's entry in the process table. General purpose registers 4 through 6 are also filled in from the process table.

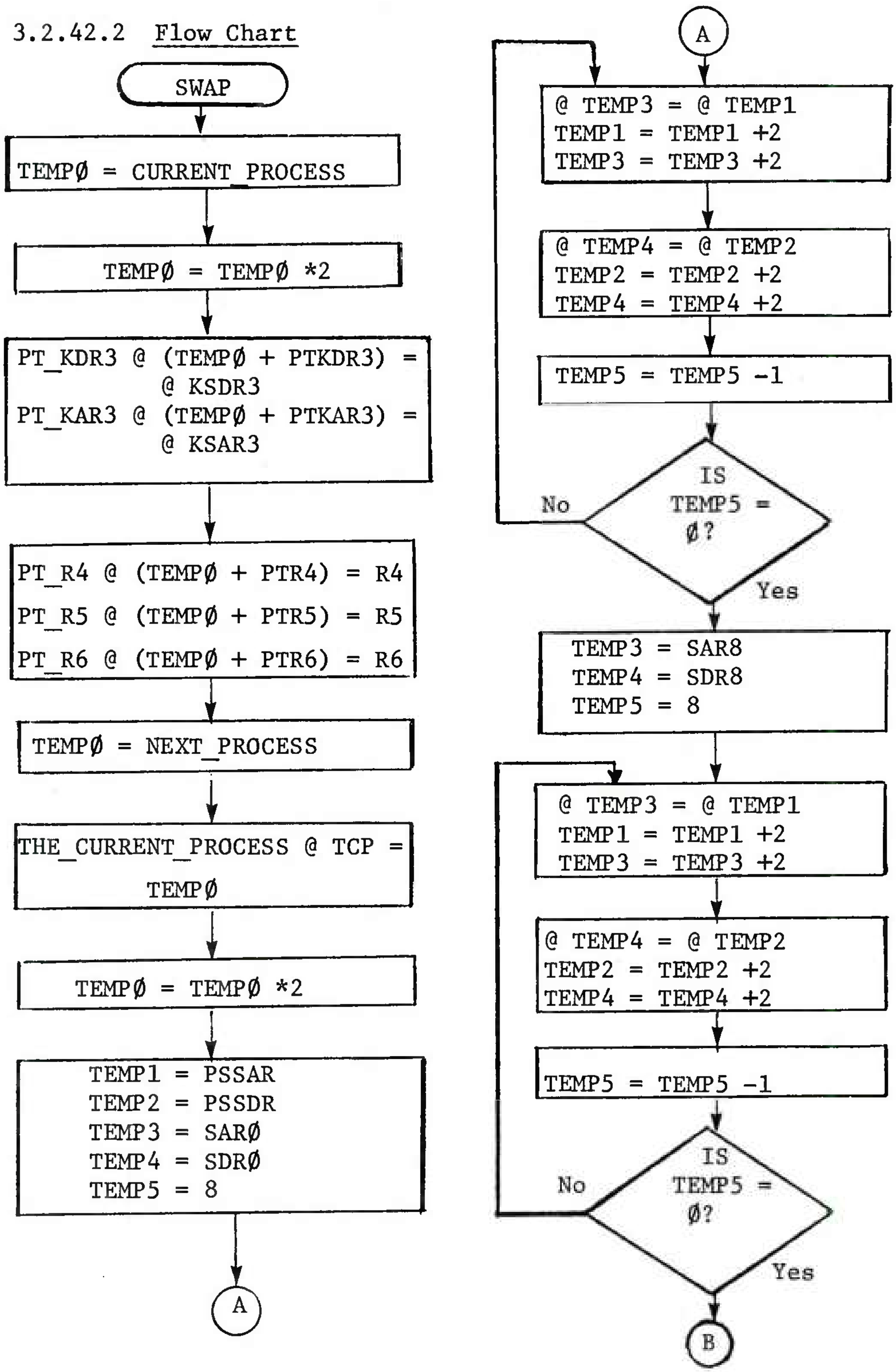
SWAP now checks if the current process is a new process. If R5 is non-zero, the process is not new, and SWAP returns to the kernel. Otherwise, it prepares to return out to the user. It clears registers Ø through 3. To R4 it assigns a pointer to the static link, and R6 gets the kernel stack pointer. The user PSW, the PC, the user R6, and the pointer to the static link are pushed onto the stack. The pointer to the static link is then popped from the stack to the static link and the user R6 is popped from the stack to R6 in supervisor mode. A return from interrupt is then executed, restoring the supervisor's RC and PSW from the kernel stack.

```
Function:  SWAP
Parameters:  SWAP(TCP_process#, process#)
Effect:
PT_KSDR3(TCP-process#) = KSDR3;
PT_KSAR3(TCP_process#) = KSAR3;
PT_R4(TCP-process#) = R4;
PT_R5(TCP_process#) = R5;
PT_R6(TCP_process#) = R6;
(∀reg#)((PS_SAR(process#, reg#) = SAR(reg#)) &
   (PS_SDR(process#, reg#) = SDR(reg#))
KSDR1 = PT_KSDR1(process#);
KSAR1 = PT_KSAR1(process#);
KSDR2 = PT_KSDR2(process#);
KSAR2 = PT_KSAR2(process#);
KSDR3 = PT_KSDR3(process#);
KSAR3 = PT_KSAR3(process#);
R4 = PT_R4(process#);
R5 = PT_R5(process#);
TCP = process#;
```

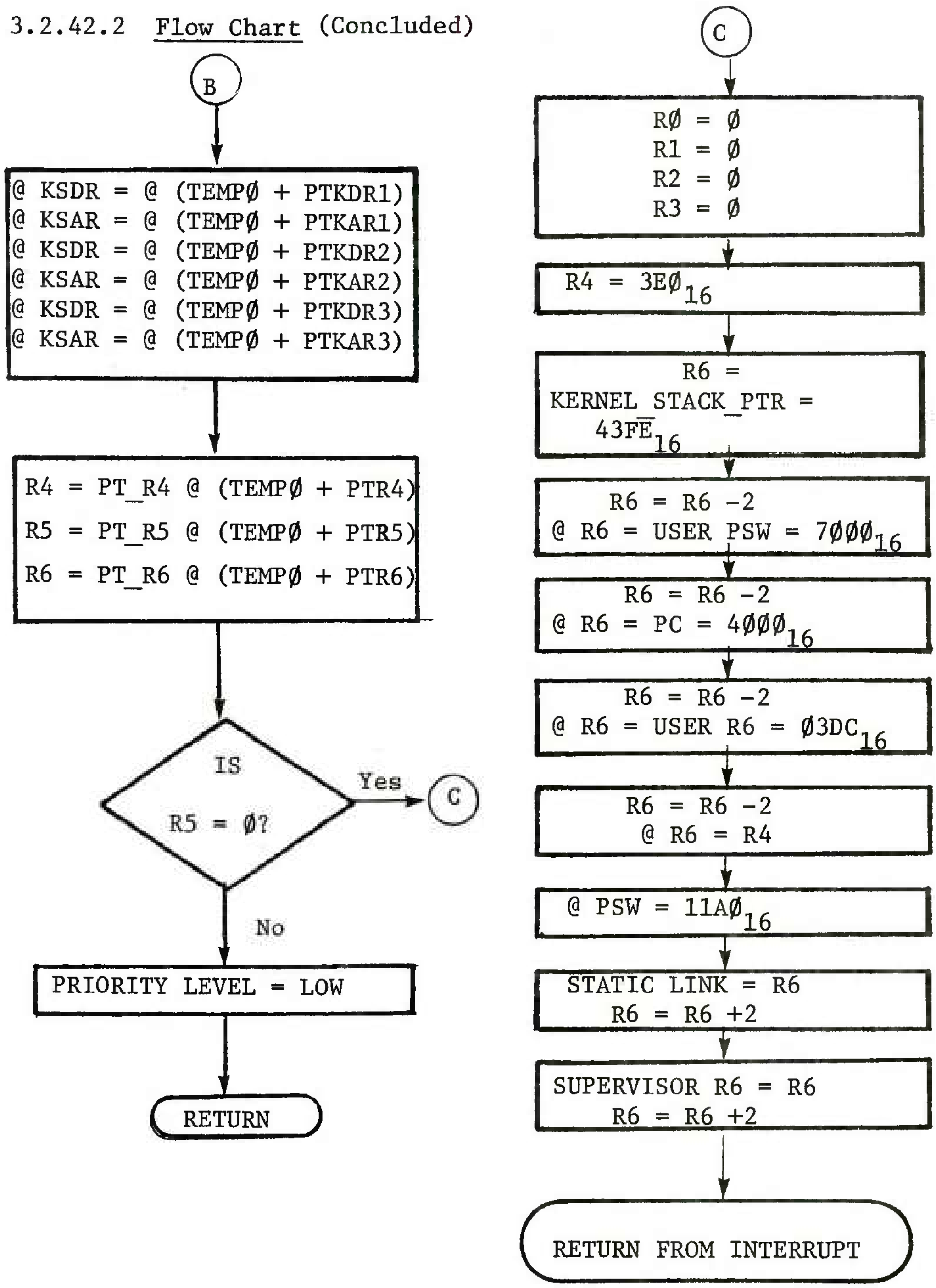
```
IF  R5 = 0
THEN:  R0 = 0;
   R1 = 0;
   R2 = 0;
   R3 = 0;
   R4 = "3E0";
   R6 = "43FE";
   SUPERV_STATIC_LINK +"3E0";
   SR6 = "3DC";
   PSE = "7000";
   PC = "4000";

END;
```

3.2.42.2 Flow Chart



3.2.42.2 Flow Chart (Concluded)



3.2.42.3 Interfaces

Refer to Figure 6, Function Call Matrix, in paragraph 3.4.

Called By	Calls
RUN	None

3.2.42.4 Data Organization

Listed below are Security Kernel data base references and constants used by the function SWAP. For data base references refer to Figure 5, Data Base Reference Matrix, in subparagraph 3.3.1. For constants refer to Table I, List of Constants, in subparagraph 3.3.2.

Data Base References

Global References	Function Parameters	Local References
PT_KSDR1	CURRENT_PROCESS	PTKDR1
PT_KSAR1	NEXT_PROCESS	PTKAR1
PT_KSDR2		PTKDR2
PT_KSAR2		PTKAR2
PT_KSDR3		PTKDR3
PT_KSAR3		PTKAR3
PT_R4		PTR4
PT_R6		PTR5
PS_SAR		PTR6
PS_SDR		KSDR1
SDR		KSAR1
SAR		KSDR2
PSW		KSAR2
		KSDR3
		KSAR3
		SARØ
		SDRØ
		SAR8
		SDR8
		TCP
		PSW

Constants

ADD	DEC
ASL	JMP
CLR	MOV

3.2.42.5 Limitations

None

3.2.42.6 Listing

```
                                                                    PAGE    1

STMT  SOURCE STATEMENT

      R0=%0
      R1=%1
      R2=%2
      R3=%3
      R4=%4
      R5=%5
      R6=%6
      PC=%7
      ;
      ;         SWAP(CURRENT_PROCESS, NEXT_PROCESS);
      ;
      ;
      ;         RELEVANT DEFINITIONS IN PT
      ;
      PTKDR1=17000
      PTKAR1=17040
      PTKDR2=17100
      PTKAR2=17140
      PTKDR3=17200
      PTKAR3=17240
      PTR4=17300
      PTR5=17340
      PTR6=17400
      ;
      ;         AND PS
      ;
      PSSDR=20000
      PSSAR=20040
      ;
      ;         HARDWARE REGS
      ;
      KSDR1=172302
      KSAR1=172342
      KSAR2=172344
      KSDR2=172304
      KSAR3=172346
      KSDR3=172306
      SAR0=172240
      SDR0=172200
      SAR8=177640
      SDR8=177600
      ;
      ;         AND FINALLY
      ;
      TCP=16544                           ;THE CURRENT PROCESS
      PSW=177776
      ;
      SWAP:     MOV    R6,R5              ;SUE
                ADD    #6,R5              ;ENTRY
                MOV    PC,(R5)+           ;SEQUENCE
                MOV    -4(R5),R0          ;CURRENT_PROCESS
                ASL    R0                 ;ARRAYS ARE OF WORDS
```

```
STMT   SOURCE STATEMENT

       ;
       ;          SAVE   KSR3     OF CURRENT PROCESS IN PT
       ;
                  MOV    @#KSDR3,PTKDR3(R0)
                  MOV    @#KSAR3,PTKAR3(R0)
       ;
       ;          SAVE   GPR4-6 IN PT
       ;
                  MOV    R4,PTR4(R0)
                  MOV    R5,PTR5(R0)
                  MOV    R6,PTR6(R0)
                  MOV    -6(R5),R0              ;NEXT_PROCESS
                  MOV    R0,@#TCP               ;SAVE
                  ASL    R0                     ;ARRAYS ARE WORDS
       ;
       ;          LOAD SR0-15 FOR NEXT PROCESS FROM ITS PS
       ;
       ;          FIRST SR0-7
       ;
                  MOV    #PSSAR,R1              ;BASE ADDRESS OF PS_SAR ARRAY
                  MOV    #PSSDR,R2              ;PS_SDR
                  MOV    #SAR0,R3               ;BASE ADDRESS OF SUPERVISOR
                  MOV    #SDR0,R4               ;SEGMENTATION REGISTERS (SR0-7)
                  MOV    #8,R5                  ;LOOP CONTROL
       SR07:      MOV    (R1)+,(R3)+
                  MOV    (R2)+,(R4)+
                  DEC    R5
                  BNE    SR07
       ;
       ;          NOW    SR8-15
       ;
                  MOV    #SAR8,R3               ;BASE ADDRESS OF USER
                  MOV    #SDR8,R4               ;SEGMENTATION REGISTERS (SR8-15)
                  MOV    #8,R5                  ;LOOP CONTROL
       SR815:     MOV    (R1)+,(R3)+
                  MOV    (R2)+,(R4)+
                  DEC    R5
                  BNE    SR815
       ;
       ;          LOAD KSR1, 2, & 3 FROM PT
       ;
                  MOV    PTKDR1(R0),@#KSDR1
                  MOV    PTKAR1(R0),@#KSAR1
                  MOV    PTKAR2(R0),@#KSAR2
                  MOV    PTKDR2(R0),@#KSDR2
                  MOV    PTKAR3(R0),@#KSAR3
                  MOV    PTKDR3(R0),@#KSDR3
       ;
       ;          AND GPR4-6
       ;
                  MOV    PTR4(R0),R4
                  MOV    PTR5(R0),R5
```

```
STMT  SOURCE STATEMENT

                MOV   PTR6(R0),R6
      ;
      ;         IF THIS IS NOT A NEW PROCESS RETURN INTO KERNEL
      ;         ELSE RETURN OUT TO USER
      ;
                MOV   R5,R5
                BEQ   NEW
                .WORD 230                  ;SPL LOW
                JMP   @-10(R5)
      ;
      ;
      NEW:      CLR   R0
                CLR   R1
                CLR   R2
                CLR   R3
                MOV   #1740,R4             ;"3E0"
                MOV   #41776,R6            ;KERNEL STACK POINTER - "43FE"
                MOV   #070000,-(R6)        ;USER PSW - CM=S, PM=U
                MOV   #040000,-(R6)        ;PC - "4000"
                MOV   #001734,-(R6)        ;USER R6
                MOV   R4,-(R6)             ;POINTER TO STATIC LINK
                MOV   #010340,@#PSW        ;FOR NEXT INSTRUCTION
                .WORD 006637 ;MTPI         ;SET STATIC LINK EQUAL TO
                .WORD 1740                 ;POINTER TO STATIC LINK
                .WORD 006606 ;MTPI R6      ;SET SUPERVISOR MODE R6
                RTI
                .END  SWAP
```

3.3 Storage Allocation

The Security Kernel consists only of its data base and its code. The Executive and Listener programs as well as the Executive stacks and root directory, while necessary for its running are not actually part of the Security Kernel. The Editor and Exerciser (which allow user interaction) while desirable are not necessary to the running of the Security Kernel; their core space may be used for storing directory and data segments. A memory map is shown in Figure 4.

Of the 64K words of core storage available, the Security Kernel data base occupies 5K words and its code occupies 8.75k words. The root directory requires Ø.5K words of core and the hardware registers 4K words of high core. System programs - namely, the Executive (including its work space), and the Listener - occupy 5.625K words of core. The Editor and Exerciser, if present, require an additional 13.5K words of storage space. This leaves about 4ØK words of core (exclusive of the Editor and Exerciser). This space is allocated consecutively in blocks of 1K bytes; that is, each segment, as it is swapped into main memory, is assigned the lowest free block of storage. No timing requirements are imposed. The only equipment constraint affecting storage allocation is that only 8K bytes may be addressed at a time; hence the Security Kernel and Editor code is stored in smaller segments.

3.3.1 Data Base Characteristics

The Security Kernel global data base structures, generally speaking, fall into two categories: need-to-know and resource management.

Included in the need-to-know category are:

a. Directories, which have a fixed part (DIR_) and a variable part (ACL_), are access matrices which describe access permissions.

b. The Active Segment Table (AST_) is the record of current access. Functions read the directories to fill in some of the information contained in the AST.

c. The Process Table (PT_) and Process Segment (PS_) contain control information about a process. The PT contains information on all segments whereas the PS contains information on a specific segment. The PT contains some information which appears in the PS.

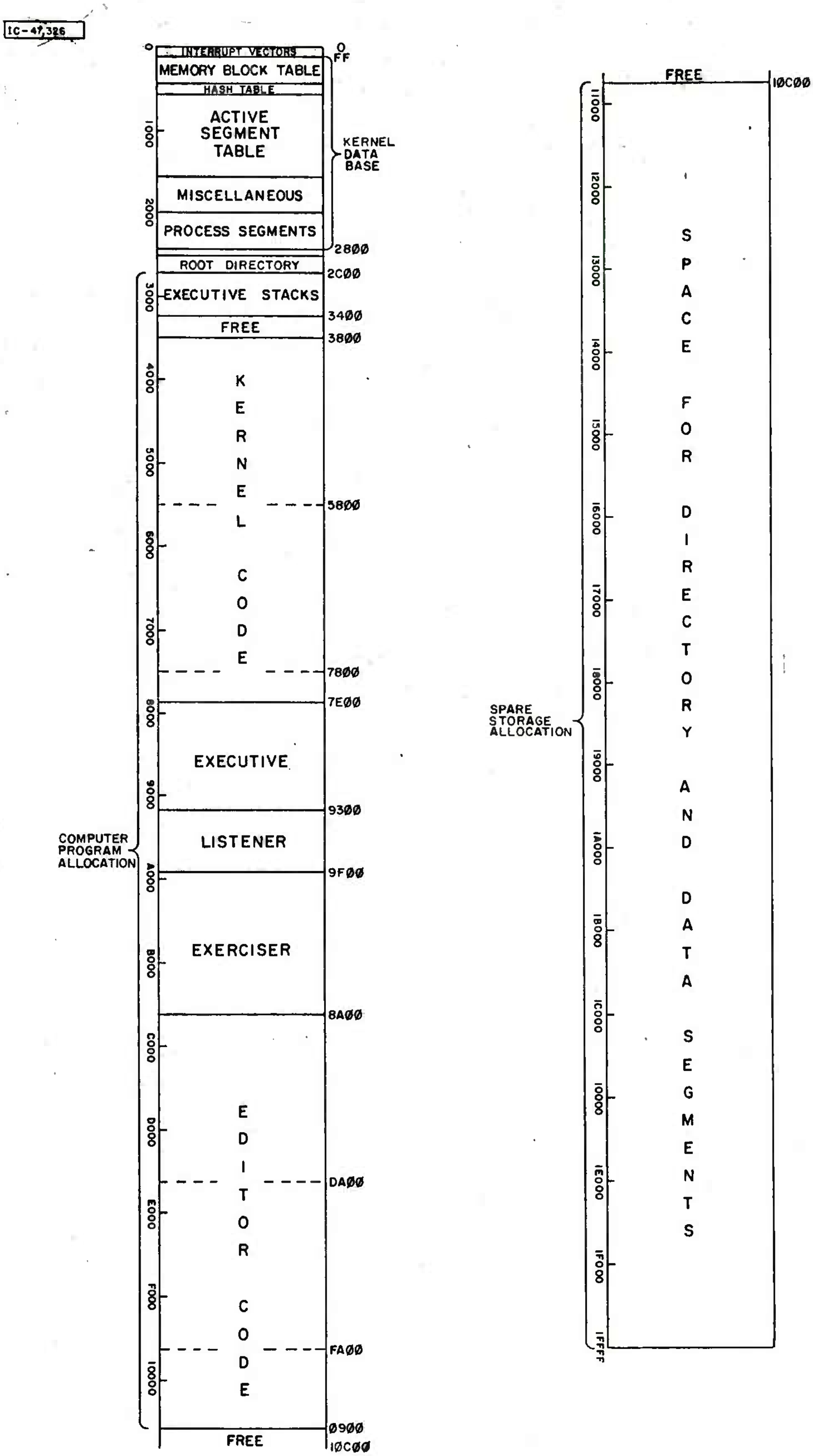


Figure 4. MEMORY MAP (ALL ADDRESSES HEXADECIMAL)

Included in the resource management category are:

a. The Memory Block Table (MBT_) which indicates the state of main memory. Entries in the MBT are Active Segment Table Entry numbers (ASTE#).

b. The Bit Map Table (MBT_) deals with disk allocation.

c. The Hash Table has but one entry (HASH_TABLE) which points to the beginning of a chain of ASTEs.

d. The Interprocess Communication Element Pool (IPC_) contains elements which are controlled by a quota mechanism. The head of the IPC queue is located in the PT.

e. Semaphore entries are arrays of Ø to 257. There is one semaphore, which is indexed by ASTE#, for each segment. The head of the chain of blocked processes on a semaphore is located in the PT.

f. Parameters is a table of user input parameters.

g. Segmentation Address Registers (SAR) and Segmentation Descriptor Registers (SDR) are register pairs containing information fully describing a segment. SARs reference the MBT for their information and the SDRs reference the AST entry SIZE.

The Data Base Reference Matrix presented as Figure 5 shows the global data base and the functions that reference each individual structure. Reading across shows which data structures are referenced by that specific function. Reading down shows which functions reference that specific data structure.

The following subparagraphs contain detailed definitions of the contents of the Security Kernel data structures. The address spaces referred to in each subparagraph are the virtual addresses of the specified segmentation register. CONTEXT KERNEL, in Section 10, contains a complete definition of the Security Kernel virtual address space. (Refer to Section 3 paragraph 4 for a description of the dynamic address translation performed by the MMU.)

DATA BASE

FUNCTION	DIRECTORIES	ACL	AST	PT	PS	MBT	BMT	HASH TABLE	IPC ELT POOL	SEMAPHORES	PARAMETERS	SDR & SAR
GATE												●
PCHECK					●						●	
CREATE	●		●									
DELETE	●											
GIVE	●	●										
RESCIND	●	●										
OUTERP			●		●					●		
OUTERV			●		●					●		
STARTP	●		●	●	●							
STOPP			●	●	●				●			
CHANGEO	●		●		●							
INITH	●		●		●							
READIR	●		●								●	●
IPCRCV				●	●				●			
IPCSEND				●	●				●			
GETW	●				●							
GETR	●				●							
ENABLE			●		●							●
WRITEDIR			●		●							
DALLOC							●					
GETDIR			●									
DELETSEG	●	●	●									
SOADD	●		●	●	●							
DFREE							●					
CONNECT	●		●		●							
DCONNECT			●		●					●		
HASH			●					●				
DSEARCH	●	●	●		●							
ACT	●		●					●		●		
DEACT			●					●				
DISABLE			●		●	●						●
PREHASH												
SWAPIN			●			●						
SWAPOUT			●			●						
INITSEG		●	●									
DISKIO												
P				●						●		
V				●						●		
SLEEP				●								
RUN			●	●		●						●
LSD			●									●
SWAP				●	●							●

1B-47,327

Figure 5 DATA BASE REFERENCE MATRIX

3.3.1.1 Directories

Functions that reference the directories either read or write the directory entries. In order to do anything with a segment, e.g., create it, delete it, give access to it, the directory that catalogues the segment must be referenced. A directory is a segment of entries that contain the attributes of some other segment. Directories have a fixed part and a variable part. The fixed part is filled in at the time the segment is CREATEd. The field names for this part of the directories begin with "DIR_". The variable part is known as the Access Control List. Its field names begin with "ACL_". The ACL is an open-ended list of names of users permitted to access the segment and implement the need-to-know protection. Directories are located in KSR3 address space (refer to Section 10, page 15).

Format of a Directory Entry
(fixed part)

A directory entry is accessed by (aste#, offset): DIR_XXX(aste#, offset)

FIELD	NO. of BITS	DESCRIPTION
DIR_TYPE	1	DIRECTORY or DATA
DIR_STATUS	1	UNINITIALIZED OR INITIALIZED
DIR_CLASS	4	security classification
DIR_CAT	16	security category set
DIR_SIZE	8	segment size in blocks
DIR_DISK	16	disk address of the segment
DIR_ACL_HEAD	8	head of the ACL chain (or Ø if the list is empty)

DIR_TYPE specifies the type attribute of the segment. Its value is either DIRECTORY (a 1-bit) or DATA (a Ø-bit).

DIR_STATUS indicates whether or not a segment has been initialized. Its value is either UNINITIALIZED (a 1-bit) or INITIALIZED (a Ø-bit).

DIR_CLASS is the classification part of the security attribute. It has values of 1 through 4; 1 equals unclassified, 2 equals confidential, 3 equals secret, and 4 equals top secret.

DIR_CAT is the category set which is the rest of the security attribute. It has possible values of Ø through 32767.

DIR_SIZE is the size of the segment in a multiple of 256 bytes. If the value of DIR_SIZE is zero, the directory entry is not being used and the value of all other fields are undefined.

DIR_ACL_HEAD is the head of a chain of ACL elements. If there are no ACL elements then DIR_ACL_HEAD is zero.

Formal of an Access Control List (ACL) Element
(variable part)

An ACL element is accessed by (aste#, acle#): ACL_XXX (aste#, acle#)

FIELD	NO. of BITS	DEFINITION
ACL_USER	14	user-id or ALL_USERS
ACL_PROJECT	8	project_id or ALL_PROJECTS
ACL_MODE	2	mode of access - WRITE, READ, or NO access
ACL_CHAIN	8	acle# of next ACL in the chain or Ø

Whenever a user is on the system the state information of his process includes a two part name identifier - user_id and project_id. An ACL element includes this two part name but either part may be replaced by a flag indicating "don't care". This special flag is represented by ALL_USERS or ALL_PROJECTS.

Each ACL element, in addition to a name, has a permitted mode of access - no access, read access, or write access. The access mode is associated with the ACL element rather than the segment itself to allow different users to have different access rights to the segment. ACL elements are ordered from most significant to least significant. Elements with a specific user_id and project-id come first, an ALL_PROJECTS element will always be last, and elements with a specific user and ALL_PROJECTS will come before an ALL_USERS, specific element.

A directory segment has 63 useable entries (numbered 1 to 63) plus an unuseable entry (entry Ø) and 127 ACL elements that are shared among all entries. The sharing mechanism employs a chain of free ACL elements - the head of this free chain is ACL_CHAIN(Ø). A directory is initialized by marking all its entries as free and placing all the ACL elements on the free chain.

All segment attributes except for DIR_STATUS and DIR_DISK are specified by users with write access to the directory and therefore have the security level of the parent directory, but the values of DIR_STATUS and DIR_DISK are a function of system wide activity.

Directories are considered to be "composite" objects. Most of the data in a directory will be at the security level of the directory but some will be at a higher level. The format of the directory is defined within the security perimeter so there is no problem in determining the security level of a particular data item. Since the segment is the smallest object to which access is controlled by the MMU, uncertified software cannot be permitted direct read access to directory segments. If uncertified software is to have read access to a directory it must be via Security Kernel functions that do the reading interpretively and are aware of the nature of the directories.

3.3.1.1.1 Functions Using Directories and Access Control List

Directories	Access Control List
CREATE	GIVE
DELETE	RESCIND
GIVE	DELETSEG
RESCIND	DESEARCH
STARTP	INITSEG
CHANGEO	
INITH	
READIR	
GETW	
GETR	
DELETSEG	
SOADD	
CONNECT	
DESEARCH	
ACT	

3.3.1.2 Active Segment Table (AST)

The Active Segment Table is a system-wide table that facilitates the main memory sharing of segments among processes. Every segment that is in the work space (WS) of one or more processes or is wired down (a permanent location in core dedicated to the segment) has an entry in the AST. The segment is identified by its aste# (AST entry #). An ASTE is composed of a number of fields. The AST is located in KSRØ address space (refer to Section 10, pages 11 and 12).

Format of an Active Segment Table Entry

An AST entry is accessed by aste#: AST_XXX(aste#)

FIELD	NO. of BITS	DESCRIPTION
AST_TYPE	1	DIRECTORY or DATA
AST_STATUS	1	UNINITIALIZED or INITIALIZED
AST_CLASS	4	security classification
AST_CAT	16	security category set
AST_SIZE	8	segment size in blocks
AST_DISK	16	disk address of the segment
AST_CHANGE	1	segment altered or unaltered while in core
AST_CPL	16	connected process list
AST_WAL	16	write access list
AST_AGE_CHAIN	16	chain for segments eligible for deactivation
AST_ADR	16	main memory address of a segment
AST_DES_COUNT	16	number of descriptors for a segment
AST_UNLOCK	1	UNLOCKED - AST_DES_COUNT: Ø
AST_SWAP_CHAIN	16	chain of segments eligible to be swapped out
AST_CHAIN	16	used by HASH functions and for ASTE chain

The head of chains is accessed by AST_XXX(Ø).

AST_TYPE, AST_STATUS, AST_CLASS, AST_CAT, AST_SIZE, and AST_DISK correspond to the similarly named fields in a directory entry. These fields in the ASTE are set by copying from the directory entry at the time the segment is activated.

AST_CHANGE indicates if the segment has been modified while enabled. A 1-bit means the segment has been altered, Ø-bit means unaltered.

AST_CPL (connected process list) indicates which processes have the active segment in their WS (read access is implied). AST_WAL (write access list) indicates which processes have write access to the segment as well. AST_CPL and AST_WAL are bit maps. Bit Ø indicates whether or not the segment is wired-down (Ø indicates not wired-down, 1 indicates wired-down). When one of the remaining bits is a 1, the corresponding process has access to the segment (AST_CPL-read access, AST_WAL-write access). When a process removes a segment from its WS, AST_CPL may become zero (no processes have the segment in their WS). This means that the segment can be deactivated making the ASTE free.

Segments that can be deactivated (as indicated by a zero AST_CPL) are kept on a chain running through AST_AGE_CHAIN. Since ASTE_WAL is not meaningful then AST_CPL is zero; ASTE_WAL and AST_AGE_CHAIN can physically overlay each other.

AST_ADR is the main memory address of a segment if it is swapped in; AST_ADR will be zero if it is swapped out. Since the beginning main memory address of a segment will always be on a 256 byte boundary, AST_ADR need not include the low order (all zero) 8 bits of the address.

AST_DES_COUNT (descriptor count) indicates the number of descriptors that exist for a segment.

Active segments that are eligible to be swapped out are kept on a chain running through the AST_SWAP_CHAIN field. When a process removes a segment from its AS, AST_DES_COUNT may go to zero. This means the segment has become unlocked and can be removed from main memory.

AST_UNLOCK indicates whether or not a segment is on the AST_SWAP_CHAIN (1-bit indicates UNLOCK, Ø-bit indicates LOCK). This one bit field allows the AST_DES_COUNT and AST_SWAP_CHAIN fields to be overlayed.

AST_CHAIN is used to chain together ASTE's that are free. The function HASH also uses the AST_CHAIN field to resolve hashing collisions.

3.3.1.2.1 Functions Using the Active Segment Table

CREATE	SOADD
OUTERP	CONNECT
OUTERV	DCONNECT
STARTP	HASH
STOPP	DSEARCH
CHANGEO	ACT
INITH	DEACT
READIR	DISABLE
ENABLE	SWAPIN
WRITEDIR	SWAPOUT
GETDIR	INITSEG
DELETSEG	RUN
	LSD

3.3.1.3 Process Table (PT)

The Process Table has an entry for each process, and each entry consists of several fields. The PT has an area to hold the basic state of all processes when they are not allocated to the processor. The Process Table is located in KSRØ address space (refer to Section 10, page 14).

Format of the Process Table

The Process Table is accessed by process#: PT_XXX(process#)

FIELD	NO. of BITS	DESCRIPTION
PT_KSDR1	16	kernel segmentation descriptor register 1
PT_KDAR1	16	kernel segmentation address register 1
PT_DSDR2	16	kernel segmentation descriptor register 2
PT_KSAR2	16	kernel segmentation address register 2
PT_KSDR3	16	kernel segmentation descriptor register 3

PT_KSAR3	16	kernel segmentation address register 3
PT_R4	16	general register 4
PT_R5	16	general register 5
PT_R6	16	general register 6
PT_CUR_CLASS	4	security classifications
PT_CUR_CAT	16	security categories
PT_KS_ASTE#	16	aste# of the kernel stack
PT_PS_ASTE#	16	aste# of the process's process segment
PT_FLAGS	2	READY, BLOCKED or INACTIVE
PT_LINK	6	chain of processes blocked on a semaphore
PT_IPC_QUEUE_HEAD	8	head of the IPC queue
PT_IPC_QUOTA	8	unused ipc element quota

The first nine entries in the PT are only used when a process becomes blocked. They are written when the process is blocked and read when the process becomes unblocked. The next four entries are fixed fields while the last four entries are variable. These fixed and variable fields of the PT are the current attributes of a process.

PT_KSDR1 and PT_DSAR1 hold the location of the process's process segment.

PT_KSDR2 and PT_KSAR2 hold the location of the process's kernel stack.

PT_KSDR3 and PT_KSAR3 hold the location of the process's current directory segment.

PT_R4, PT_R5, and PT_R6 are general registers whose contents are important to the SUE language. They act as accumulators, stack pointers and temporaries. Register 6 has the special function of the processor stack pointer.

PT_CUR_CLASS is the security classification of the process.

PT_CUR_CAT is the security category set of the process.

PT_KS_ASTE# keeps the aste# of the process's kernel stack.

PT_PS_ASTE# keeps the aste# of the process's segment which contains more information about the process.

PT_FLAGS indicates the execution state of a process. Its value is READY, BLOCKED or INACTIVE.

PT_LINK is used to chain together processes that are blocked on the same semaphore.

PT_IPC_QUOTA_HEAD is the beginning of a chain of interprocess communication messages sent to the process. Its value indicates one of three possible states: (1) there are messages that have been sent and not yet read by the process; (2) there are no messages that have been sent to the process and not yet read by the process; and (3) the process has been blocked because it wants to read another message and none is available.

PT_IPC_QUOTA is a number of interprocess communication objects currently available to the user for receiving messages from other processes.

3.3.1.3.1 Functions Using the Process Table

STARTP	P
STOPP	V
IPCRCV	SLEEP
IPCSEND	RUN
SOADD	SWAP

3.3.1.4 Process Segment (PS)

There is a Process Segment (main memory segment) for each process. The PS is created at initialization time and along with the appropriate PT entry, holds information on the state of the process. The Process Segment is located in KSR1 address space (refer to Section 10, pages 14 and 15).

Format of a Process Segment

Process Segments are accessed by process#: PS_XXX(process#)

FIELD	NO. of BITS	DESCRIPTION
PS_CURRENT PROCESS	8	process#
PS_PROCESS_MASK	16	bit mask

PS_PROCESS_NOTMASK	16	bit mask
PS_USER_ID	14	user identification
PS_PROJECT_ID	8	project identification
PS_CUR_CLASS	4	security classification
PS_CUR_CAT	16	security category
PS_MEM_QUOTA	8	unused main memory quota
PS_SDR	16 x 16 array	save area for user and supervisor domain segmentation registers
PS_SAR	16 x 16 array	save area for user and supervisor domain segmentation registers
PS_SEG	15 x 32 array	definition of process's address space

PS_CURRENT_PROCESS is the number of the process associated with the PS.

PS_PROCESS_MASK and PS_PROCESS_NOTMASK are used in accessing the ACL_CPL and ACL_WAL. They are the process# expressed by a 16-bit field; the value of PS_PROCESS_MASK is expressed as $2^{15-\text{process\#}}$, whereas the value of PS_PROCESS_NOTMASK is the complement of PS_PROCESS_MASK. MASK is all zero except for the bit indicating the process#. NOTMASK is all ones except for the bit indicating the process#.

PS_USER_ID and PS_PROJECT_ID identify the user associated with the process.

PS_CUR_CLASS and PS_CUR_CAT define the classification and category which is the current security level of the process.

PS_MEM_QUOTA is the amount of main memory allocated to the process for its AS but not currently in use.

PS_SDR and PS_SAR are two arrays that form the save area for the 8 supervisor (Ø-7) and 8 user (7-15) segmentation registers.

PS_SEG is used for mapping segment numbers (seg#'s) into aste#'s. Every segment in a users WS has an aste#, but the aste# cannot be available to the user because it is a function of system wide activity. Therefore, when a process has the kernel move

a segment into its WS, the kernel returns a seg# which the process subsequently uses to identify the segment.

3.3.1.4.1 Function Using the Process Segment

PCHECK	GETW
OUTERP	GETR
OUTERV	ENABLE
STARTP	WRITEDIR
STOPP	SOADD
CHANGEO	CONNECT
INITH	DCONNECT
IPCRCV	DSEARCH
IPCSEND	DISABLE
	SWAP

3.3.1.5 Memory Block Table (MBT)

The Memory Block Table is a structure used to indicate the state of main memory. Contiguous blocks (256 bytes per block) can be concatenated to form main memory segments of any multiple block size. There is an entry in the MBT for each block with segments represented by several concatenated entries. The Memory Block Table is located in KSRØ address space (refer to Section 10, page 11).

Format of the Memory Block Table

The MBT is accessed by block#: MBT_XXX(block#)

FIELD	NO. of BITS	DESCRIPTION
MBT_FLAGS	2	FREE, ALLOCATED, or CONCATENATED
MBT_SIZE	8	size of the area in blocks
MBT_CHAIN	14	chain of free blocks
MBT_ASTE#	8	aste# of the virtual memory segment in the block

If a block is the first in a segment, MBT_FLAGS for that block is either FREE or ALLOCATED; otherwise it is CONCATENATED. The remaining fields are not meaningful for CONCATENATED blocks.

MBT_SIZE is the number of blocks in a segment.

If a block is FREE, MBT_CHAIN is the block# of the next segment in the free chain or the initial address of high core if this is the end of the chain. (A block# is the address of the first byte in a block with the 8 low order Ø bits removed.)

If the block is ALLOCATED, MBT_ASTE is the aste# of the segment bound to it.

3.3.1.5.1 Functions Using the Memory Block Table

DISABLE
SWAPIN
SWAPOUT
RUN

3.3.1.6 Bit Maps Table (BMT)

Bit maps are used exclusively by the disk allocation functions. There is an allocated area on the disk for each of the three segment sizes (the initial implementation of the Security Kernel uses only size 2), a bit map for each area, and a Bit Map Table for each bit map. The Bit Map Table is a table of tables and is located in KSRØ address space (refer to Section 10, page 13).

Format of the Bit Map Table

The SUE language functions pass the virtual address of the Bit Map Table associated with the size to the PAL-11 routines which will access the corresponding bit map.

FIELD	NO. of BITS	DESCRIPTION
BMT_SIZE1	64	segment SIZE1 bit map table
BMT_SIZE2	64	segment SIZE2 bit map table
BMT_SIZE3	64	segment SIZE3 bit map table
BIT_MAP2	512	one bit per 1K byte segment
BIT_MAP1	32	reserved for future use
BIT_MAP3	32	reserved for future use

BMT_SIZEn is the segment size n Bit Map Table. The Bit Map Table contains the start address (first 16 bits) and end address (next 16 bits) of the bit map, the base address of the disk area (next 16 bits) and a shift register (last 16 bits).

BIT_MAP2, in the initial implementation, is a map of 512 bits. The entire disk is allocated to segment SIZE2, that is, 512 1K byte segments. Each bit in the bit map corresponds to a segment on the disk. When a segment is allocated space on the disk, the corresponding bit in the bit map is set. When the space is freed the bit is cleared.

In future implementations the disk will be separated into three areas, one area for each of the three different sized segments.

3.3.1.6.1 Functions Using the Bit Map Table

DALLOC
DFREE

3.3.1.7 Hash Table

The Hash Table has only one field which is the disk address of a specific process. The Hash Table is located in KSRØ address space (refer to Section 10, page 11).

Format of Hash Table

A Hash Table entry is accessed by hash value: HASH_TABLE(HASH_VALUE)

FIELD	NO. of BITS	DESCRIPTION
HASH_TABLE	16	pointer into the AST_CHAIN

HASH_TABLE could be thought of as AST_CHAIN_HEAD as it is actually the first non-free element in the AST_CHAIN.

3.3.1.7.1 Functions Using the Hash Table

HASH
ACT
DEACT

3.3.1.8 Interprocess Communication (IPC) Element Pool

The IPC Element Pool chains messages waiting to be received. The pool is a shared resource of 127 elements controlled by a quota mechanism (each receiving process is restricted to 8 message elements). The IPC Element Pool is located in KSRØ address space (refer to Section 10, page 11).

Format of an IPC Element Pool

An IPC Element is accessed by index: IPC_XXX(INDEX)

FIELD	NO. of BITS	DESCRIPTION
IPC_LINK	8	chained IPC entry #
IPC_PROCESS#	8	sending process # and domain indicator
IPC_DATA	16	message

IPC_LINK is the chained IPC entry #, that is, the number of the next oldest element in the receiving process's chain of waiting elements.

IPC_PROCESS# contains the number of the sending process and a 1-bit domain indicator.

IPC_DATA contains the message being sent.

3.3.1.8.1 Functions Using the IPC Element Pool

STOPP
IPCRCV
IPCSEND

3.3.1.9 Semaphores

Semaphore entries are arrays of Ø to 257. The first 255 semaphores are associated with active segment, that is, SMFR# = ASTE#. The kernel semaphore equals 256 and the disk semaphore equals 257. Semaphores are located in KSRØ address space (refer to Section 10, page 13).

Format of Semaphores

A semaphore entry is accessed by aste#: SMFR_XXX(aste#)

FIELD	NO. of BITS	DEFINITION
SMFR_COUNT	16	P decrements the count V increments the count
SMFR_POINTER	16	points to chains of blocked processes

SMFR_COUNT is decremented when a P is performed; when a V is performed it is incremented. The boundries of SMFR_COUNT are -128 to 127.

SMFR_POINTER is an entry number into the PT_LINK which is the head of the chain of blocked processes (1 chain for each SMFR #).

3.3.1.9.1 Functions Using Semaphores

OUTERP
OUTERV
DCONNECT
ACT
P
V

3.3.1.10 Parameters

Parameters are a special case as they are actually part of the supervisor's data base. The kernel does, however, access this information when needed.

User input parameters are passed to the Security Kernel by placing them in fixed locations on the supervisor's stack. The Security Kernel accesses the supervisor's stack through kernel segmentation register 3. Only those parameters required by the requested function are entered by the user and accessed by the Security Kernel. Parameters are located in KSR3 address space (refer to Section 10, page 15).

Format of Parameter Entries

FIELD	NO. of BITS	DESCRIPTION
FUNCTION_CODE_APARM	16	identifies a requested function
SEG#_APARM	16	identifies as active segment
OFFSET_APARM	16	identifies a directory entry
CLASS_APARM	16	classification
CAT_APARM	16	category set
SEG_TYPE_APARM	16	DATA or DIRECTORY
SIZE_APARM	16	size of a segment in blocks
MODE_APARM	16	WRITE, READ or NO access
USER_APARM	16	user_id

PROJECT_APARM	16	project_id
REG#_APARM	16	identifies a segmentation register
PROCESS#_APARM	16	identifies a process
MESSAGE_APARM	16	IPC message
KRC	16	return code
KRC2	16	return code 2

FUNCTION_CODE_APARM is a numerical tag from 1 to 20 which identifies a function.

SEG#_APARM is the segment number of a segment in a process's address space (WS).

OFFSET_APARM is the identification of an entry within a directory.

CLASS_APARM is the classification part of the security attribute.

CAT_APARM is the category set which is the rest of the security attribute.

SEG_TYPE_APARM identifies the segment as either DATA or DIRECTORY.

SIZE_APARM is the size of the segment in 256 byte blocks.

MODE_APARM is the mode of access which is either WRITE, READ, or NO access.

USER_APARM and PROJECT_APARM are the user and project identification.

REG#_APARM is the identification of a segmentation register.

PROCESS#_APARM is a numerical identification of a process. For the IPCRCV function this field identifies the process# plus domain.

MESSAGE_APARM is an interprocess communication message.

KRC is a per process return code.

KRC2 is used by the IPCRCV function exclusively and is assigned the value of the IPC message.

3.3.1.10.1 Functions Using Parameters

PCHECK
READIR

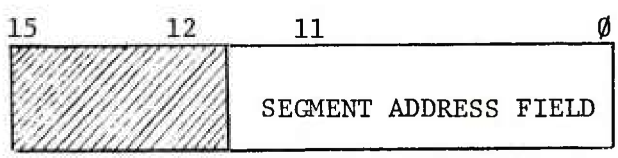
3.3.1.11 Segmentation Registers

The kernel functions that access information contained in the segmentation registers are:

GATE
READIR
ENABLE
DISABLE
RUN
LSD
SWAP

3.3.1.11.1 Segment Address Registers (SAR)

The Segment Address Register is a base address register. It contains the base address of the segment in the form of a 12-bit Segment Address Field (SAF). Bits 15-12 of the SAR are not implemented. The SAF is interpreted in address calculations as a multiplier of 32, i.e., the lowest 6 bits are assumed to be Ø. The bit stored in bit Ø of the SAF becomes bit 6 of the segment base address, bit 1 of the SAF, bit 7 of the segment base address, etc. Thus, bit 11 of the SAF becomes bit 17 of the segment base addressed.



SEGMENT ADDRESS REGISTER FORMAT

3.3.1.11.2 Segment Descriptor Registers (SDR)

The Segment Descriptor Register (SDR) contains segment length, access control, and written into fields.

15	14 8	7	6	5 4	3	2 Ø
	SEGMENT LENGTH FIELD	A	W		E D	ACF

SEGMENT DESCRIPTOR REGISTER FORMAT

3.3.1.11.2.1 Access Control Field (ACF)

The ACF is a 3-bit field (occupying bits 2-Ø of the SDR) which describes the access rights to this specific segment. The access codes specify the manner in which a segment may be accessed and whether or not a given access should result in a trap or an abort of the current process. A memory reference that causes an abort is terminated immediately. That is, an aborted "read" reference does not obtain any data from the location and an aborted "write" reference does not change the data in the location. A reference that causes a trap is completed.

Access Control Field Keys

AFC	KEY	DESCRIPTION	FUNCTION
000	0	non-resident	abort all process
001	1	read-only and trap	trap on read abort any attempt to write on this segment
010	2	resident read only	abort any attempt to write on this segment
011	3	illegal	reserved for future use
100	4	resident read/write and trap	memory management trap upon completion of a read or write
101	5	resident read/write and trap when write	memory management trap upon completion of write
110	6	resident read/write	read or write allowed - no trap or abort
111	7	illegal	reserved for future use

3.3.1.11.2.2 Written Into (W)

The W bit (occupying bit 6) indicates whether the segment has been written into since it was swapped into main memory. A W bit of 1 is affirmative indicating that the user has modified the segment and that it must be saved in its current form. A W bit of Ø indicates that the segment has not been modified and that it need not be written onto disk to be saved. The W bit is automatically cleared when either the SDR or SAR of a segment is written into.

3.3.1.11.2.3 Segment Length Field (SLF)

The 7-bit SLF (occupying bits 14-8) specifies the authorized length of the segment in 32-word blocks. A segment consists of at least one and at most 128 blocks, and occupies contiguous core location.

3.3.1.11.2.4 Attention (A) and Expansion Direction (ED)

The A bit (bit 7) and ED bit (bit 3) are not currently referenced by the Security Kernel.

3.3.2 Constants and Macros

The following two tables list all constant and macros used by the Security Kernel. Table I lists the constants contained in CONTEXT NOFORN and CONTEXT KERNEL. A description of each constant and its value is included. Table II lists the macros contained in CONTEXT NOFORN, CONTEXT KERNEL and DATA GATE. The effect of each macro and its parameters is included. The listings of CONTEXT NOFORM, CONTEXT KERNEL and DATA GATE can be found in Section 10, pages 2 through 19.

Table I

List of Constants

Values of the following are in hexidecimal unless otherwise indicated.

CONSTANT	VALUE	COMMENTS
MEM_SIZE	511_{10}	128K bytes = 512 256-byte blocks
MBT_FLAGS_MASK*	C000	memory block table
MBT_CHAIN_MASK	3FFF	FLAGS, CHAIN, and ASTE all share a word

*MASK is used as an "anding" operation for selecting a bit from a bit table

Table I (Continued)

END_BLOCK#	3E0	
ALLOCATED	0	setting of memory
CONCATENATED	4000	block table flag
FREE_MEM	8000	field
RESERVED_MEM	C000	
ASTE#_MIN	1_{10}	range of active
ASTE#_MAX	255_{10}	segment table entries
AST_TYPE_MASK	80	active segment table entries
AST_STATUS_MASK	40	TYPE, STATUS, CHANGE, UNLOCK,
AST_STATUS_NOTMASK	BF	and CLASS share a byte
AST_CHANGE_MASK	20	
AST_UNLOCK_MASK	10	
AST_LOCK_MASK	EF	
AST_CLASS_MASK	OF	
AST_TYPE_DIRECTORY	AST_TYPE_MASK	definition of active segment table TYPE entry
AST_CHANGE	AST_CHANGE_MASK	definition of active seg-
AST_UNCHANGED_MASK	DF	ment table change bit
AST_UNITIALIZED	AST_STATUS_MASK	definition of active segment table STATUS entry
AST_UNLOCK_FLAG	AST_UNLOCK_MASK	definition of active segment table UNLOCK entry
WIRED_DOWN_MASK	8000	Bit 0 of the connected pro-
WIRED_DOWN_NOTMASK	7FFF	cess list is wired down
WIRED_DOWN	WIRED_DOWN_MASK	bit
ROOT_ASTE#	1_{10}	active segment table entry ROOT constant
IPC_MAX	127_{10}	number of elements in the interprocess communications pool
IPC_QUOTA	8_{10}	receiving processes are restricted to 8 message elements

Table I (Continued)

BMT_SIZE1_ADR BMT_SIZE2_ADR BMT_SIZE3_ADR	1B00 1B08 1B10	definition of bit map table SIZE entry address
KERNEL_SMFR DISK_SMFR SMFR_MAX	256_{10} 257_{10} DISK_SMFR	definition of KERNEL and DISK semaphores
SEG#_FLAG OFFSET_FLAG CLASS_FLAG REG_FLAG PROCESS#_FLAG MODE_FLAG	8000 4000 2000 1000 0800 0400	definition of parameter flags
PT_KSDR1_ADR PT_KSDR2_ADR	1E00 1E40	segmentation descriptor and address registers must be separated by 20_{16}
PT_FLAGS_MASK PT_LINK_MASKS	CO 3F	process table entries FLAGS and LINK share a byte. LINK is only meaningful when FLAGS = BLOCKED
BLOCKED READY INACTIVE	00 40 80	definition of flag field
IPC_WAIT	FF	a process is waiting for a message
PS_KSR_ADR PS_SDR_ADR	F4C2 2000	segmentation descriptor and address registers must be separated by 20_{16}
SEG_FLAG	8000	changed to aste# when the segment is allocated
SEG_MASK	7FFE	used to mask out SEG_FLAG
MEM_QUOTA EXEC_MEM_QUOTA	24 7F	processes are restricted to 9K words of memory except the executive which is virtually unrestricted

Table I (Continued)

STACK_KSR_ADR	F4C4	definition of the stack segmentation register
DIR_KSR_ADR	F4C6	definition of the directories segmentation register
ACL_MAX	127_{10}	number of active control list elements to be shared
DIR_TYPE_MASK	80	directory entries
DIR_STATUS_MASK	40	TYPE, STATUS, and CLASS
DIR_STATUS_NOTMASK	BF	share a byte
DIR_CLASS_MASK	0F	
DIR_CLASS_NOTMASK	F0	
DIR_TYPE_DIRECTORY	80	definition of directory
DIR_UNINITIALIZED	40	entries TYPE and STATUS
ACL_MODE_MASK	C000	access control list entries
ACL_USER_MASK	3FFF	MODE and USER share a word
REG_CONSTANT	578	definition of accessing
P_REG#_MAX	587	supervisor and user seg-
CROSS_REG#	7_{10}	mentation registers
SDR_ADR	F480	supervisor SRO = 3F480 user SRO - 3FF80 REG_CONSTANT = ((3FF80 - 3F480)/2)-8=578_{16}
SDR_WRITE_ACCESS	0006	definition of
SDR_READ_ACCESS	0002	descriptor register
SDR_CHANGE_MASK	0040	fields
SDR_CHANGED	SDR_CHANGE_MASK	
PREV_MODE_MASK	3000	on kernel entry
PREV_MODE_SUPERV	1000	call is ignored if not made from supervisor mode

Table I (Continued)

DISK_WRITE	0043	disk commands
DISK_READ	0045	

The following are hardware instructions. Their values are in octal unless otherwise indicated.

ADD	0006	add instruction
ASL	0063	arithmetic shift left
ASR	0062	arithmetic shift right
BCC	103(1)0*	branch on carry clear instruc.
BVS	102(1)1	branch on overflow clear instruc.
CLR	0050	clear instruction
DEC	0053	decrement instruction
INC	0052	increment instruction
JMP	00001	jump instruction
MOV	0001	move source instruction (word)
MOVB	0011	move source instruction (bytes)
NEG	0054	negate instruction
SUB	0016	substract source instruction
SWAB	0003	swap bytes
TRAP	104400	trap instruction
MUL	070100	multiply instruction R1 = R0 * R1
DIV	071002	divide instruction R0 = R0R1/R2
ASHR1	072127	Shift arithmetically (Register 1)
ASHR0R3	072003	shift arithmetically (Register 0, Register 3)
ASHR1R3	072103	shift arithmetically (Register 1, Register 3)
XORL0	074100	Exclusive OR
MFPIR6	006506	move from previous instruction space
MTPIR6	006606	move to previous instruction space
SPLHIGH	000237	set priority level high
SPLLOW	000230	set priority level low

* A (1) indicates that the value following is in binary.

Table I (Continued)

The following values are in decimal unless otherwise indicated

CREATE_FUNCTION_CODE	1	
DELETE_FUNCTION_CODE	2	
GIVE_FUNCTION_CODE	3	
RESCIND_FUNCTION_CODE	4	
GETW_FUNCTION_CODE	5	
GETR_FUNCTION_CODE	6	
RELEASE_FUNCTION_CODE	7	
ENABLE_FUNCTION_CODE	8	
DISABLE_FUNCTION_CODE	9	
P_FUNCTION_CODE	10	
V_FUNCTION_CODE	11	
T_FUNCTION_CODE	12	
IPCSEND_FUNCTION_CODE	13	
IPCRCV_FUNCTION_CODE	14	
STARTP_FUNCTION_CODE	15	
STOPP_FUNCTION_CODE	16	
CHANGEO_FUNCTION_CODE	17	
PROCID_FUNCTION_CODE	18	
INITH_FUNCTION_CODE	19	
READIR_FUNCTION_CODE	20	
FUNCTION_CODE_MIN	1	range of function code
FUNCTION_CODE_MAX	20	
SEG#_MIN	1	range of segment numbers
SEG#_MAX	31	
ROOT_SEG#	1	
OFFSET_MIN	1	range of offsets
OFFSET_MAX	63	
PDD_OFFSET	1	process directory directory
IOD_OFFSET	2	input/output directory
CD_OFFSET	3	code directory
FMS_OFFSET	4	file management system
UNCLASSIFIED	1	definition of class
CONFIDENTIAL	2	
SECRET	3	
TOP_SECRET	4	
CLASS_MIN	1	range of class
CLASS_MAX	4	

Table I (Continued)

SEG_TYPE_DIRECTORY	80_{16}	definition of segment type
SEG_TYPE_DATA	00_{16}	
SIZE1	1	256 bytes
SIZE2	4	1K bytes
SIZE3	16	4K bytes
NO_ACCESS	0	definition of mode
READ$EXECUTE_ACCESS	4000_{16}	
WRITE$READ$EXCUTE_ACCESS	$C000_{16}$	
ALL_USERS	3FFF	definition of user and project
ALL_PROJECTS	$7F_{16}$	
SYSTEM_PROJECT	1	
REG#_MIN	0	range of register numbers
REG#_MAX	15	
PROCESS#_MIN	1	definition of process numbers
PROCESS#_MAX	7	
PROCESS#_2MAX	14	
EXEC_PROCESS#	1	
TTY_PROCESS#	2	
DECW_PROCESS#	3	
SCOPE1_PROCESS#	4	
SCOPE2_PROCESS#	5	
USER_PROCESS#_MIN	2	
USER_PROCESS#_MAX	5	
PROCESS#_MASK	$7F_{16}$	definition of IPCRCV PROCESS# (PROCESS# plus DOMAIN)
DOMAIN_MASK	80_{16}	
KERNEL_DOMAIN	DOMAIN_MASK	
OF_FLAG	$FFFF_{16}$	definition of kernel return code
ERR_FLAG	$FFFE_{16}$	
SEVERE_FLAG	$FFFD_{16}$	
TRUE	1_{10}	
FALSE	0_{10}	
MAXIMUM_INTEGER	32767_{10}	
MAX_NEG_INTEGER	-32768_{10}	
CV_MAX_LEN	72_{10}	character varying maximum length
CARRIAGE_RETURN	D_{16}	
LINE_FEED	25_{16}	

Table I (Concluded)

LOW_CHARACTER	80_{16}	
HIGH_CHARACTER	07_{16}	
NEW_LINE	N	
END_OF_FILE	*	
BS_CHAR	@	back space
CANCEL_CHAR	LINE_FEED	

Table II

List of Macros

MACRO NAME	PARAMETERS	EFFECT
MULTIPLY	OP1, OP2, PRODUCT, FLAG	Places the product of OP1 and OP2 in PRODUCT and sets FLAG if result is less than -2^{15} or greater than or equal to 2^{15}.
DIVIDE	DIVIDEND, DIVISOR, QUOTIENT, FLAG	Places the result of DIVIDEND/DIVISOR in quotient and sets FLAG if dividing by zero is attempted.
MUDOLO	DIVIDEND, DIVISOR, REMAINDER, FLAG	Finds the REMAINDER and sets FLAG in the event of an overflow.
KCREATE	SEG#, OFFSET, CLASS, CAT, SEG_TYPE_SIZE, RC	Calls kernel function CREATE and sets RC to KERNEL_RC.
KDELETE	SEG#, OFFSET, RC	Calls kernel function DELETE and sets RC.

Table II (Continued)

KGIVE	SEG#, OFFSET, USER, USER, PROJECT, RC	Calls kernel function GIVE and sets RC.
KRESCIND	SEG#, OFFSET, USER, PROJECT, RC	Calls kernel function RESCIND and sets RC.
KGETW	SEG#, OFFSET, RC	Calls kernel function GETW and sets RC.
KRELEASE	SEG#	Calls kernel function DCONNECT
KENABLE	SEG#, REG#, RC	Calls kernel function ENABLE and sets RC.
KDISABLE	REG#	Calls kernel function DISABLE and sets RC.
KP	SEG#, RC	Calls kernel function OUTERP and sets RC.
KV	SEG#, RC	Calls kernel function OUTERV and sets RC.
KT	SEG#, RC	Enters kernel to read semaphore and sets RC.
KIPCSEND	PROCESS#, MESSAGE	Calls kernel function IPCSEND.
KIPCRCV	PROCESS#, MESSAGE	Calls kernel function IPCRCV to read MESSAGE and sending process#.
KSTARTP	USER_ID, PROJECT_ID, CLASS, CAT, PROCESS#, PROC_OFFSET, RC	Calls kernel function STARTP and sets RC.
KSTOPP	______	Calls kernel function STOPP.
KCHANGEO	SEG#, OFFSET, CLASS, CAT, RC	Calls kernel function CHANGEO and sets RC.

Table II (Concluded)

KPROCID	PROCESS#	Enters the kernel to find the process's PROCESS#.
KINITH	SEG#, OFFSET, ASTE#, RC	Calls kernel function INITH and sets RC.
KREADIR	SEG#, OFFSET, CLASS, CAT, SEG_TYPE, SIZE, RC	Calls kernel function READIR to find the CLASS, CAT, SEG_TYPE, and SIZE attributes of directory entry SEG#, OFFSET.
KERNEL_ENTRY	———	Pushes user registers R0 to R6 onto kernel stack.
KERNEL_EXIT	———	Restores user registers R0 to R6 from kernel stack.

3.4 Security Kernel Function Call Matrix

The Function Call Matrix presented in Figure 6 lists the forty-two functions of the Security Kernel. Reading across the matrix shows which functions are called by a specific function. Reading down shows which functions a specific function calls. The matrix also shows whether a specific function is externally or internally callable, the number of functions it calls, and the number of functions that call it. Eighteen of the functions are externally callable. The remaining twenty-four are called either directly or indirectly by the externally callable functions. The non-callable functions are invisible outside the kernel domain and deal basically with the management of the PDP-11/45's physical resources. The externally callable functions may be invoked by any process operating in supervisor domain with the exception of STARTP, CHANGEO, and INITH which may be called by one trustworthy process, the Executive Process.

IC- 47,328

CALLED FUNCTION

CALLING FUNCTION	GATE	PCHECK	CREATE	DELETE	GIVE	RESCIND	OUTERP	OUTERV	STARTP	STOPP	CHANGEO	INITH	READIR	IPCRCV	IPCSEND	GETW	GETR	ENABLE	WRITEDIR	DALLOC	GETDIR	DELETSEG	SOADD	DFREE	CONNECT	DCONNECT	HASH	DSEARCH	ACT	DEACT	DISABLE	PREHASH	SWAPIN	SWAPOUT	INITSEG	DISKIO	P	V	SLEEP	RUN	LSD	SWAP	NC	NF	E/I	
GATE		•																																				•					2	0	I	GATE
PCHECK			•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•								•					•						•	•					20	1	I	PCHECK
CREATE																			•	•																							2	1	E	CREATE
DELETE																			•		•	•					•		•														5	1	E	DELETE
GIVE																			•				•																				2	1	E	GIVE
RESCIND																			•				•																				2	1	E	RESCIND
OUTERP																																					•						1	1	E	OUTERP
OUTERV																																						•					1	1	E	OUTERV
STARTP																•	•	•								•	•														•		6	1	E	STARTP
STOPP															•											•												•	•				4	1	E	STOPP
CHANGEO																			•								•																2	1	E	CHANGEO
INITH																			•																								1	1	E	INITH
READIR																																	•								•		2	1	E	READIR
IPCRCV																																					•	•	•				3	1	E	IPCRCV
IPCSEND																																											0	2	E	IPCSEND
GETW																									•			•															2	2	E	GETW
GETR																									•			•															2	2	E	GETR
ENABLE																																	•								•		2	2	E	ENABLE
WRITEDIR																																	•								•		2	6	I	WRITEDIR
DALLOC																																											0	1	I	DALLOC
GETDIR																																	•								•		2	1	I	GETDIR
DELETSEG																							•	•			•			•													4	1	I	DELETSEG
SOADD																										•	•	•													•		4	3	I	SOADD
DFREE																																											0	1	I	DFREE
CONNECT																											•		•														2	2	I	CONNECT
DCONNECT																															•												1	4	E	DCONNECT
HASH																																											0	6	I	HASH
DSEARCH																																	•								•		2	3	I	DSEARCH
ACT																														•		•											2	2	I	ACT
DEACT																																•	•	•									3	2	I	DEACT
DISABLE																																											0	2	E	DISABLE
PREHASH																																											0	2	I	PREHASH
SWAPIN																																		•	•	•	•						4	6	I	SWAPIN
SWAPOUT																																				•	•						2	2	I	SWAPOUT
INITSEG																																									•		1	1	I	INITSEG
DISKIO																																											0	2	I	DISKIO
P																																					•	•	•				3	6	I	P
V																																											0	6	I	V
SLEEP																																								•			1	3	I	SLEEP
RUN																																									•	•	2	1	I	RUN
LSD																																											0	9	I	LSD
SWAP																																											0	1	I	SWAP

<u>LEGEND</u>

NC – NUMBER OF FUNCTIONS CALLED BY THIS FUNCTION
NF – NUMBER OF FUNCTIONS THAT CALL THIS FUNCTION
E/I – EXTERNALLY OR INTERNALLY CALLABLE

Figure 6 KERNEL FUNCTION CALL MATRIX

3.4.1 Program Interrupts

Program interrupts occur when a new process requests the service of the CPU. Before servicing the new process, however, the CPU finishes executing the instruction it is working on. The interrupt is then handled by the invocation of the KERNEL_ENTRY macro, which causes the contents of the current process's registers to be saved. The PC and PSW of the interrupt vector now become the new process's PC and PSW. The function V is then called to increment the count on the semaphore associated with the new process's I/O segment. Macro KERNEL_EXIT is then invoked which causes the general purpose registers, the PC and the PSW to be restored with what they contained before the interrupt.

3.4.2 Subprogram Referencing

The following figures depict the calling flow of the Security Kernel. To facilitate readability the Security Kernel has been broken down into three levels; the entry point GATE (Figure 7), the eighteen externally callable functions (Figure 8 through 21), and the internal function SWAPIN (Figure 22).

GATE

↓

PCHECK

→

CREATE → See Figure 8
DELETE → See Figure 9
GIVE → See Figure 10
RESCIND → See Figure 10
GETW → See Figure 11
GETR → See Figure 11
DCONNECT → See Figure 12
ENABLE → See Figure 13
DISABLE (calls no other component)
OUTERP → See Figure 14
OUTERV → See Figure 15
IPCSEND (calls no other component)
IPCRCV → See Figure 16
STARTP → See Figure 17
STOPP → See Figure 18
CHANGEO → See Figure 19
INITH → See Figure 20
READIR → See Figure 21

Figure 7. GATE Call Diagram

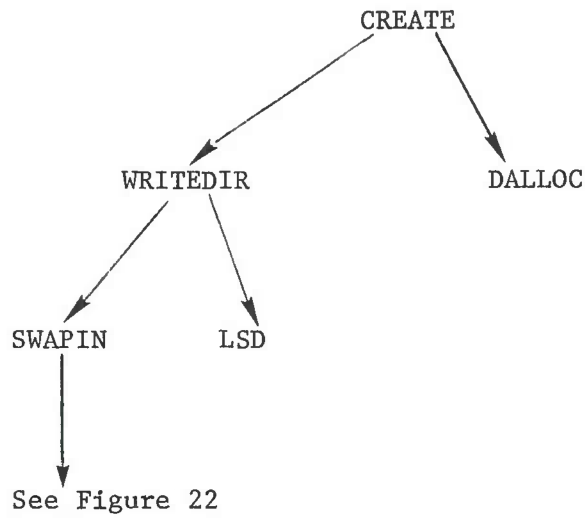


Figure 8. CREATE Call Diagram

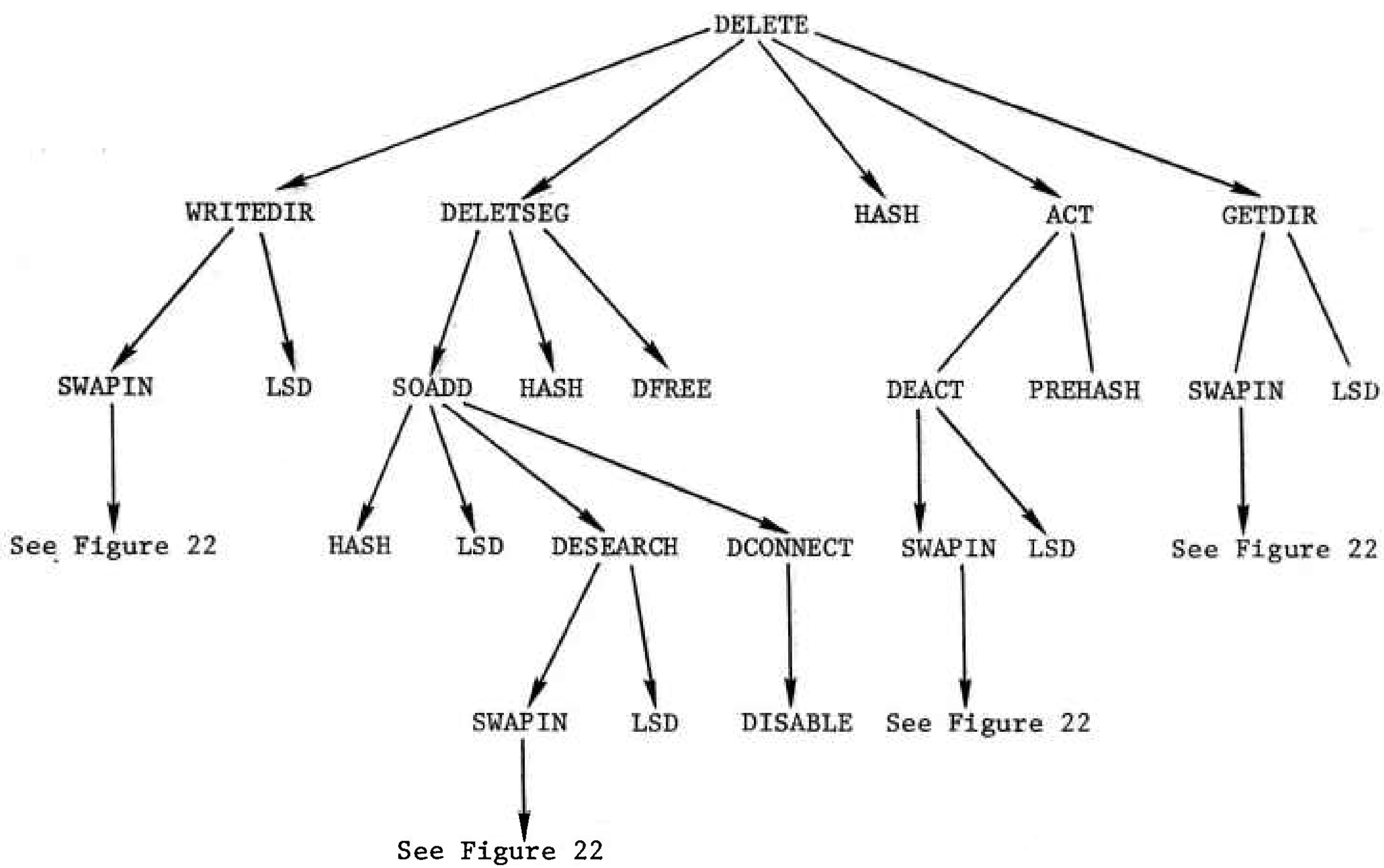


Figure 9. DELETE Call Diagram

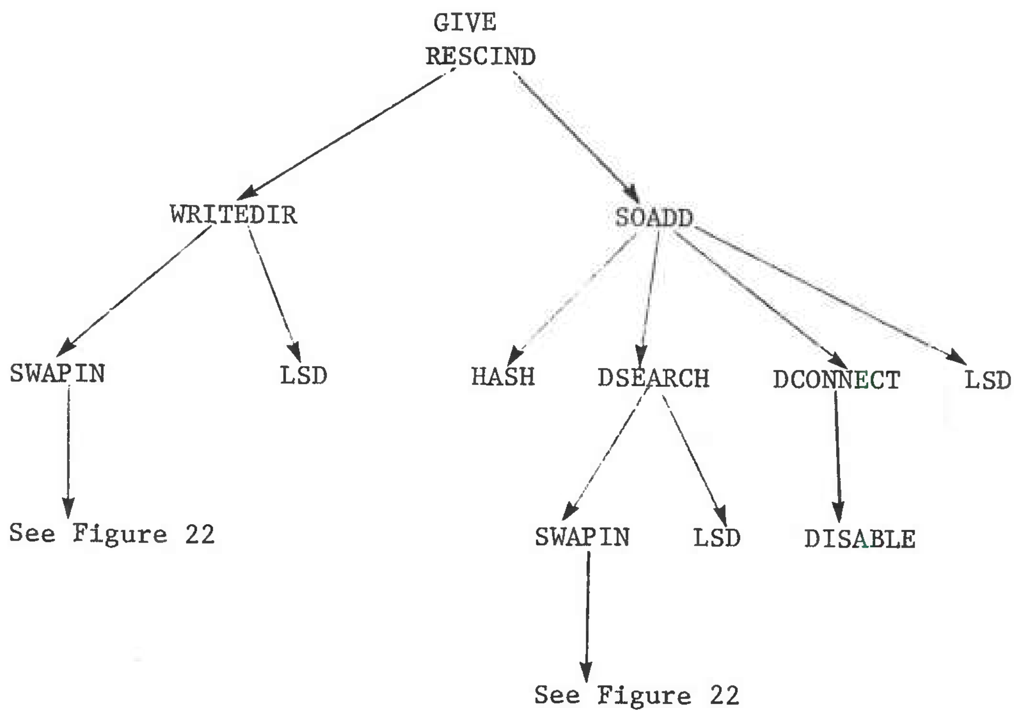


Figure 10. GIVE and RESCIND Call Diagram

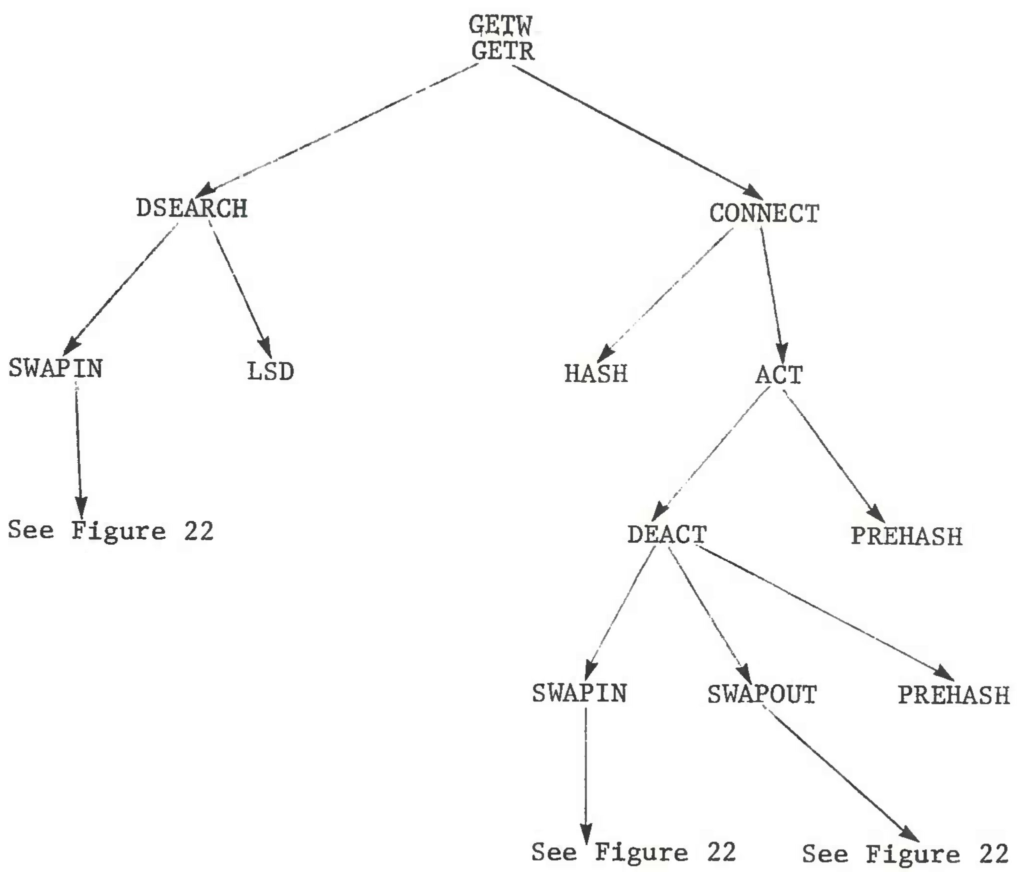


Figure 11. GETW and GETR Call Diagram

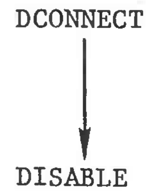


Figure 12. DCONNECT Call Diagram

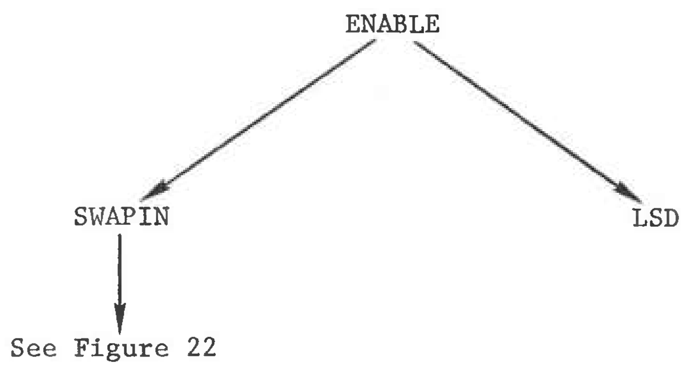


Figure 13. ENABLE Call Diagram

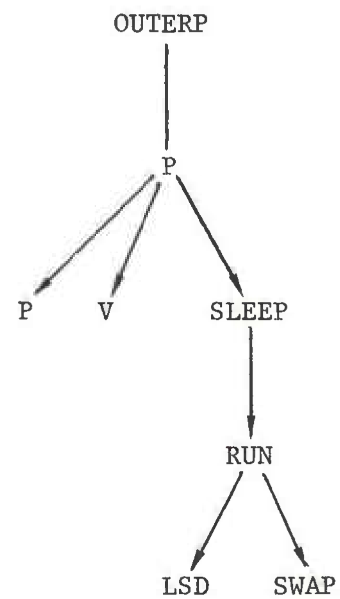


Figure 14. OUTERP Call Diagram

Figure 15. OUTERV Call Diagram

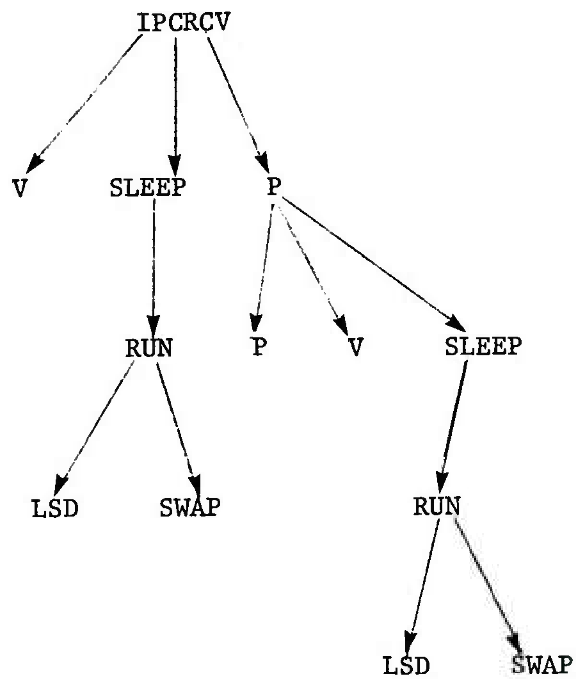


Figure 16. IPCRCV Call Diagram

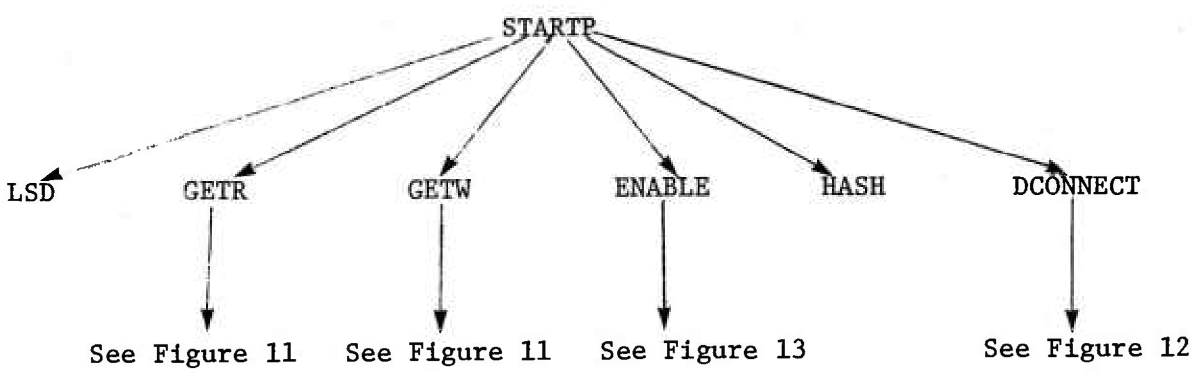


Figure 17. STARTP Call Diagram

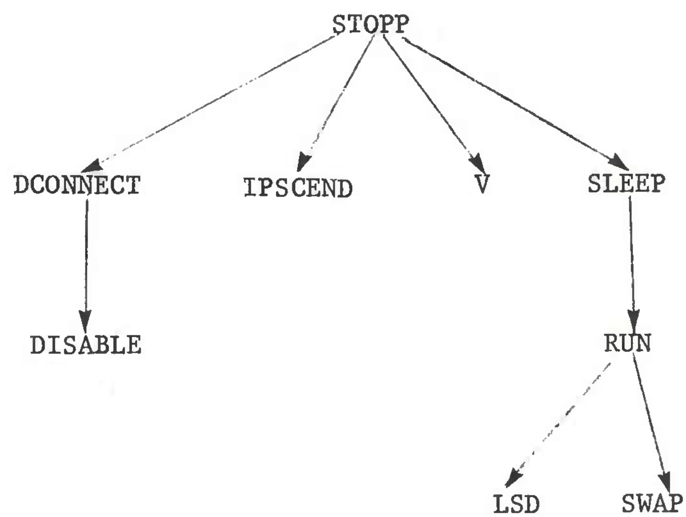


Figure 18. STOPP Call Diagram

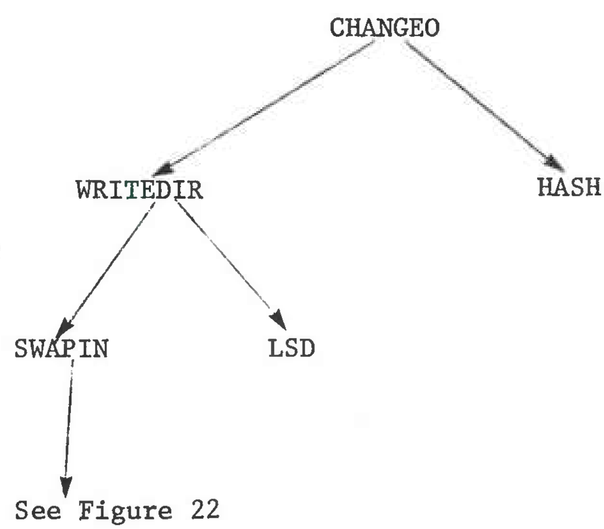


Figure 19. CHANGEO Call Diagram

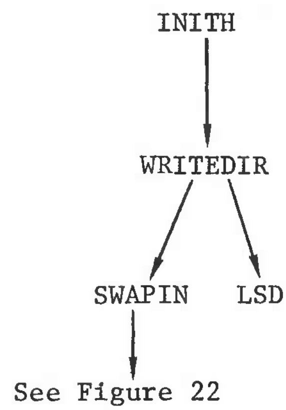


Figure 20. INITH Call Diagram

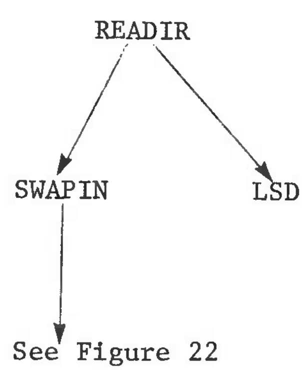


Figure 21. READIR Call Diagram

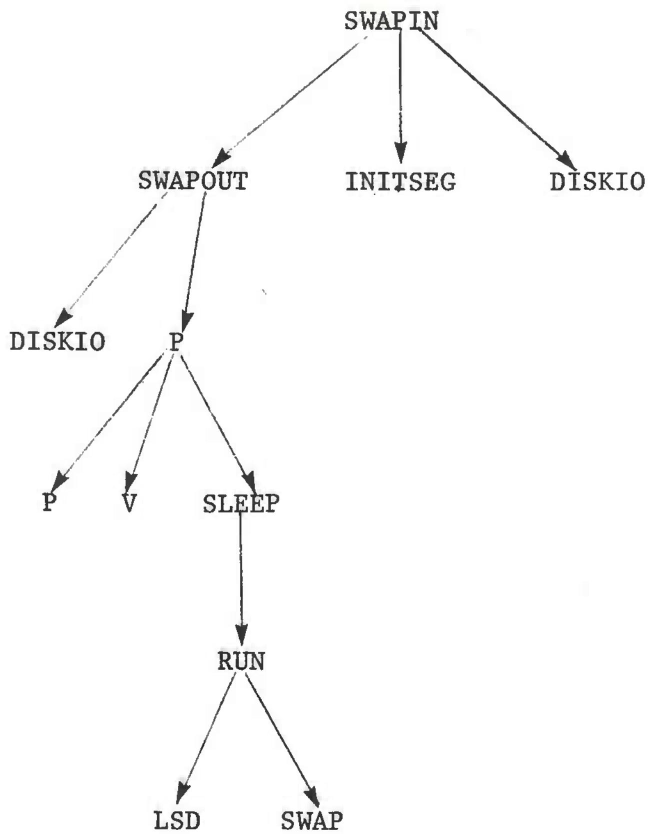


Figure 22. SWAPIN Call Diagram

3.4.3 Special Control Features

This paragraph briefly describes three system programs which may be run in conjunction with the Security Kernel.

3.4.3.1 STARTUP

STARTUP sets up a basic environment in which to run the Security Kernel. It reads the Kernel, Executive, and Listener code from magnetic tape into core. It then initializes the root directory, activates the Executive's working space, and invokes the Executive.

3.4.3.2 EXECUTIVE

EXECUTIVE creates basic directories (the process directory directory, the I/O directory and the code directory) subordinate to the root. It places code segments into the code directory, I/O segments into the I/O directory, and creates its own process directory off the process directory directory. It then establishes a Listener program for each connected terminal. Two subprograms of the Executive, PSTART and PSTOP, are called to initiate and terminate Listener and user processes.

3.4.3.3 LISTENER

LISTENER accepts a user's correct start command, then loads the user, project, class and category into a message segment to be read by the Executive, which will start a user process with the specified security characteristics. There is a Listener process for each user station.

4. QUALITY ASSURANCE

4.1 Validation Criteria

Department of Defense Regulation DoD 5200.1-R (ref., paragraph 2.1b this specification) governs the Classification, Downgrading, Declassification, and Safeguarding of Classified Information pursuant to DoD Directive 5200.1 (ref., paragraph 2.1a), the Department of Defense Information Security Program. This program and Regulation addresses the problem of protection of official information relating to National Security, to the extent and for such period as is necessary. The Regulation establishes the bases for identification of information to be protected; establishes a progressive system for classification, downgrading and declassification; prescribes safeguarding policies and procedures to be followed; and establishes a monitoring system to insure the effectiveness of the Information Security Program throughout the Department of Defense.

DoD 5200.1-R provides the following definition of information: "knowledge that can be communicated in any form". It also provides the following policy with respect to certain official information: "To protect against actions hostile to the United States,...it is essential that such official information... be given only limited dissemination". To implement this policy, it states that such information be designated as needing protection, i.e., that it be classified. To further aid in implementing this policy the regulation states that "the dissemination of classified information orally, in writing, or by other means, shall be limited to those persons whose official duties require knowledge or possession (need-to-know) thereof" and, more specifically, no person shall be eligible for access to classified information unless a determination has been made as to his trustworthiness, i.e., unless he has been given the requisite level of security clearance.

4.1.1 Information Security Model

In order to implement a computer system providing the requisite security of official information from any possibility of compromise, it is necessary that that system behave in the machine domain in precise and complete correspondence with the regulations and intent of the DoD Information Security Program. The concepts of regulation DoD 5200.1-R (of people, information, and limiting access to information), provide the basis for representing the DoD Information Security Program in the form of an Information Security Model. This model will be validated, by the approving authority, to be a precise and sufficient algorithmic statement of the functions corresponding to the requirements and definitions of DoD 5200.1-R. Upon validation,

this model shall be the controlling criterion against which the acceptability of the Security Kernel Computer Program Product (described in Section 3 of this specification) will be measured for validation.

4.1.1.1 Elements of the Information Security Model

The Information Security Model, which is a precise algorithmic statement of security functions, consists of four elements: subjects, objects, an access control mechanism, and an authorization data base. The model describes the security requirements to be satisfied by subjects (people or processes) for accessing objects, in any specifically identified mode, under control of the Security Kernel. In the meaning of this paragraph, objects can be files, messages, buffers, terminals, I/O devices, etc. Objects can be accessed by subjects only in accordance with the compromise prevention requirements stipulated by the access control mechanism of the Security Kernel.

4.1.2 Validation Tests and Demonstrations

A specific program of demonstrations and tests shall be performed to verify that the functionality of the Security Kernel Computer Program Product (SKCPP) precisely and completely corresponds with the concepts of the Information Security Model, and also that no functionality of the SKCPP fails to correspond precisely and completely with one or more concepts of the model. These demonstrations and tests will take the form of rigorous, logically sound proofs of correspondence, and may be performed in sequential steps of validation which form a step-by-step validation correspondence proof chain stretching between the executable machine code (the least abstract representation of the Security Kernel) and the Information Security Model (the most abstract representation of the Security Kernel). Such a validation chain is discussed in paragraph 4.1.2.1 below.

4.1.2.1 The Validation (Correspondence Proof) Chain

The process of validation of the Security Kernel has as its goal the clear and rigorous proof that the conceptual solution of the real-world problem of prevention of compromise of information security, as represented by the Information Security Model, has been precisely implemented on the particular hardware/software system that will deal with that real-world problem.

Specifically, it is required that the functionality of the hardware/software system that consists of the binary language representation of the Security Kernel, correctly installed and operating in a

DEC PDP-11/45 computer with Memory Management Unit, be rigorously proved to completely and exclusively correspond to the functionality described by the Information Security Model.

The said validation goal requires that all aspects of the proofs be thoroughly rigorous and that they be clearly documentable. Unfortunately, the formal language and semantics in which the SKCPP is expressed are not directly comparable with the logical structure of the Information Security Model. This fact would make direct correspondence between these two representations impracticable to prove and document. Instead, a multi-step, continuous chain of correspondence proofs, similar in form to that illustrated in Figure 23, shall be performed. In this generalized validation chain design begins with the most abstract representation (the Information Security Model) and proceeds in steps through more concrete representations until it reaches the most concrete form, the useable system (hardware/software binary machine language) representation of the Security Kernel. At each link in the chain, it is required that any and all state transformations that are possible in the less abstract (more concrete) representation be proved to correspond exactly and completely with expected state transformations in the more abstract representation.

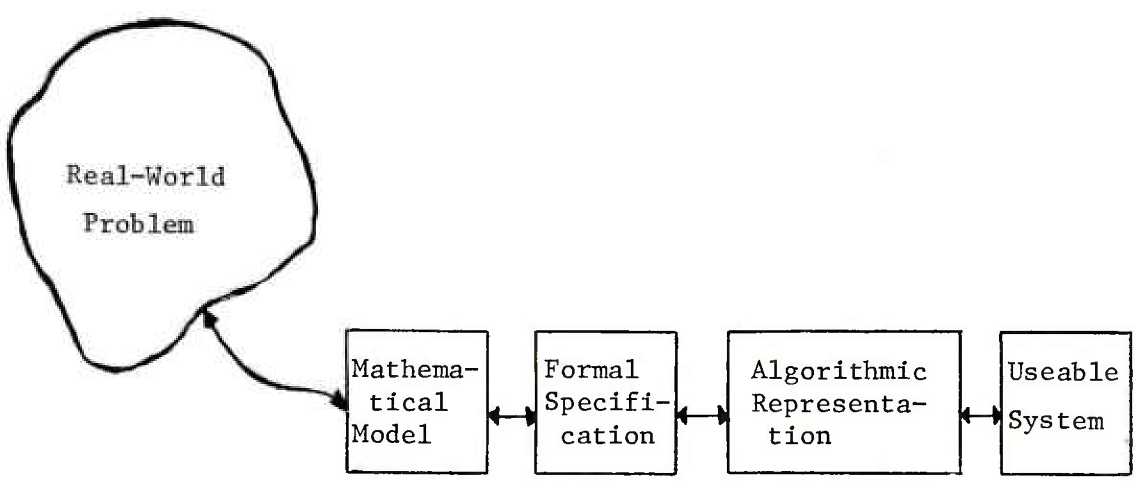


Figure 23. The Validation Chain

4.1.2.2 Validation Chain Components

Figure 23 illustrates four kinds of representations, at differing levels of abstraction, that may be used to implement validating the proof of correspondence between the useable hardware/software binary "machine language" representation of the Security Kernel and the "mathematical model" of computer security of information. The four representations, leading from the most abstract to the most concrete, are:

a. mathematical model,

b. formal specification,

c. algorithmic representation, and

d. useable system.

As used here, the "mathematical model" refers to the model described in reference 2.2a; the "formal specification" refers to the "Parnas specification" for each computer program component which appears as paragraphs 3.2.n.1 in Section 3 of this specification; and the "algorithmic representation" is the SKCPP SUE language and PAL-11 representations described in Section 3 of this specification.

The illustration implies that correspondence proofs will follow the path through the identical representations to those used in design to show that less abstract corresponds with the more abstract representation. However, the actual proofs of correspondence for validation need not follow the identical path through the representations that were used in development of the SKCPP, provided that the correspondence proofs constitute a rigorously continuous chain of proofs that prove the correspondence between the useable machine language Security Kernel and the Information Security Model.

The correspondence proof chain may, for example, follow a chain of proofs such as the following:

a. Useable hardware/software binary machine language representation.

b. PAL-11 assembler language representation.

c. SUE language representation.

d. PARNAS formal specification.

e. Information Security Model.

5. PREPARATION FOR DELIVERY

This section states the requirements incumbent upon the Contractor for preparing the Security Kernel Computer Program Product (SKCPP) for delivery to the Government and for insuring the integrity and security of the product as delivered for validation and final delivery.

5.1 Preparation of Useable Machine Language SKCPP

Prior to the start of validation tests and demonstrations, the Contractor shall take all the actions necessary to prepare, produce and protect the integrity of a precise binary machine language representation of the SKCPP described in the SUE and the PAL-11 languages in Section 3 of this specification. The Contractor shall use the (combined SUE and PAL-11) high-order language SKCPP representation of this specification as the source level code, in a security controlled IBM 360 environment where the Project SUE system and the PDP-11 cross assembler both execute, to compile the SKCPP machine language load module on 9-track magnetic tape media for transfer to the PDP-11/45. The Contractor shall protect the integrity of this preparation process and the resulting machine language code media as required by paragraph 5.1.1 below.

5.1.1 Protection of SKCPP Integrity

The Contractor shall take all the action necessary to insure, protect, and preserve the accuracy and integrity of the SKCPP binary machine language load module for the PDP-11/45. These actions shall include but not be limited to:

a. Protection of the integrity of the source level code through supervisory control and monitoring by a specific hierarchical group of persons, referred to hereinafter as the Kernel Control Group (KCG), selected by the Contractor and the Government and approved by the authority designated by the Government agency responsible for the SKCPP procurement. The Contractor shall provide the KCG with free access to monitor and review the correctness of the source level code and all the processes employed in compiling the SKCPP binary machine language load module in the tape media. The KCG shall have complete configuration control for the SKCPP. The Contractor shall be responsible for submitting to the KCG, in advance of implementation, any contemplated modification whatsoever to the SKCPP source level code, the compilation environment and procedures or the contents of

the load module. No changes shall be made to any of these entities without prior approval by the KCG.

b. Protection of the physical media, in which the binary machine language version of the Security Kernel resides, from any possibility of unauthorized alteration; this protection shall be commensurate with the level of protection required by the highest level of information security classification and special access categories for which the system, in which the said Security Kernel will be installed, must be cleared.

c. Those actions necessary to allow disclosing the contents of and information describing the SKCPP binary machine language load module as if it were unclassified, while protecting its security against modification to the extent required by (b) above.

d. The Contractor shall maintain, at all times subsequent to its initial compilation, a duplicate copy of the machine language version of the SKCPP, and shall also maintain an up-to-date history of any and all modifications that occur to the original copy of the SKCPP.

e. Whenever modification occurs to the binary machine language SKCPP, the duplicate copy shall be modified to maintain its identity with the original copy, and the SKCPP must immediately be purged from the computer until its revalidation has been completed and approved.

f. Revalidation after modification of the SKCPP binary machine language media shall be performed and documented following procedures meeting the requirements of Sections 4.1.2, 4.1.2.1, and 4.1.2.2 of this specification.

6. NOTES

The following notes are provided informally to assist the potential Security Kernel user.

NOTES ON THE MITRE PDP-11/45 PROTOTYPE
SECURITY KERNEL

The PDP-11/45 kernel distribution consists of two 9 track magnetic tapes and documentation. The tapes are referenced as the PDP-11 tape and the OS/360 tape. These notes document the contents of the two tapes and will serve as a guide to additional documentation:

a. ESD-TR-75-69, "The Design and Specification of a Security Kernel for the PDP-11/45", by W. L. Schiller, May 1975.

b. Memo, "Project 7070 versus IBM OS/TSO and the Project SUE System Language", by J. A. Larkins, August, 1975. This memo was originally intended for use by MITRE Project 7070 personnel, so that some of the information it contains is relevant only to the MITRE IBM system.

c. Notes on "Using the Kernel Exerciser".

d. Notes on "Using the ALTER Program".

e. PDP-11/45 Configuration Chart.

f. PDPTAPE1 job output and tape dump.

g. OSTAPE2 job output.

h. SUE compilation listings for

   - STARTUP
   - KERNEL
   - EXEC
   - LISTENER
   - EXERCISE

i. PAL assembly listings for

- VDUMP
- BOOT
- DALLOC
- DFREE
- LSD
- DISKIO
- SWAP

For information on the SUE Compiler itself, contact Dr. R. C. Holt, Computer Systems Research Group, University of Toronto.

PDP-11 TAPE

The PDP-11 tape can be bootloaded onto a PDP-11 by the firmware Bootstrap Loader (MR11-DB). It contains object code of the Security Kernel, a program which initializes the Kernel, programs that run in conjunction with the Kernel, and a simple test program. The tape distributed has been successfully loaded onto MITRE's PDP-11/45, but it is configuration dependent. Physically, the tape consists of eight records followed by a file mark. The following paragraphs describe the contents of each record.

Record 1

Record 1 is a short record of all zeros. It is only on the tape because MITRE's bootstrap loader skips the first record and loads the second.

Record 2

Record 2 is BOOT, a short program written in PAL-11. BOOT is loaded by the bootstrap loader starting at location Ø and then control is passed to it (at Ø). BOOT loads record 3 starting at location 8ØØØ and then passes control to it. Since record 3 is a SUE program, BOOT like most other programs on the tape is cognizant of the SUE runtime environment and initializes general purpose registers 4 and 6 appropriately.

Record 3

Record 3 contains STARTUP. STARTUP does the initialization that is necessary before the first Security Kernel function can be invoked - it puts the system into Z_o, the initial secure state. STARTUP initializes the Security Kernel's data bases, reads in the rest of the

tape, and transforms itself into the executive process. STARTUP does a few things that may be unnecessarily complex. The main memory in which STARTUP runs is allocated to the Process Segments, to the two executive stacks and to the ROOT directory. The stacks are not accessed until STARTUP is finished, but it must initialize the Process Segments and the ROOT while it is running. The final transition into the executive process is also a little complex.

Record 4

Record 4 contains VDUMP, a simple, stand-alone debugging program written in PAL-11. It uses hexadecimal notation and displays main memory locations when started. Since it runs with the MMU, it can access any area in core, given the corresponding descriptors. VDUMP is invoked by manually branching to location 34ØØ (hexadecimal).

Record 5

Record 5 contains the Security Kernel.

Record 6

Record 6 contains EXEC, the code that the executive process executes. EXEC runs on the Security Kernel but the executive process has special privileges - it is the root process and is the only process that can create new processes. It is also the only trusted subject in the system.

EXEC has two phases - a one-time-only phase and a steady-state phase. In the one-time-only phase it establishes the initial hierarchy by creating some directories and putting code and I/O segments into them. In the steady-state EXEC responds to user logon and logoff requests by starting processes that run the EXERCISER or the LISTENER.

Record 7

Record 7 contains the LISTENER, a program that the executive runs in a process for each free terminal. The LISTENER responds to user logon requests at its process's terminal. If the request is valid the LISTENER destroys itself, an event that is detectable by the executive. The response of the executive is to start a user process running the EXERCISER for the terminal. Communication between LISTENER processes and the executive is through shared data segments. The format of the logon request is:

START <user-id> <project-id> <class> [<cat>]

user-id must be a decimal number greater than 7 and less than 32767.

project-id must be a decimal number greater than 1 and less than 127.

class must be T, S, C, or U.

cat is an optional parameter that can be any 16 bit decimal number.

The LISTENER does not perform any type of user authentication.

Record 8

Record 8 contains the EXERCISER, a test program that permits a user at a terminal to invoke arbitrary Kernel functions with arbitrary input parameters. Further details are given in the enclosed outline, "Using the Kernel Exerciser".

GENERATING THE PDP-11 TAPE

The PDP-11 tape is generated (on the S/360) with the following JCL and control cards:

```
           //   EXEC WLSTAPE
           BOOT
           STARTUP       1
           VDUMP
           KERNEL        4
           EXEC          1
           LISTENER      2
           EXERCISE     10
columns:   1
```

The catalogued procedure WLSTAPE and the program it invokes, RELOCATE, are included on the OS/360 tape. RELOCATE writes record 1 onto the tape and then starts reading the control cards. Each control card gives the member name of a program in a library and a relocation indicator. (Ø is assumed if the relocation indicator is omitted.) For each control card-RELOCATE reads the program into S/360 core with a LOAD macro, undoes the relocation performed by OS, redoes the relocation for the PDP-11, performs a byte reversal (because the PDP-11 puts the even byte in the right hand side of the halfword), and dumps the program onto the tape. RELOCATE is able to redo the relocation because it knows the format of object programs produced by the SUE-11 compiler - only VCON's need be relocated, and all VCONs begin at a fixed point relative to the beginning of each procedure. RELOCATE requires that the entry point of each program

be a relative Ø. This requirement is easily satisfied by following a simple convention for compilations and link edits. The relocation factor for the PDP-11 is the relocation indicator x2 (hexadecimal). RELOCATE is not idiot proof - an improperly formed program could cause it to loop.

Since it is likely that you will have to modify our PDP-11 software, the OS/360 tape includes the bulk of our program development environment, in addition to the PDP-11 source and object. The document entitled "Project 7070 versus IBM 370 OS/TSO and The Project SUE System Language", should serve as a guide to our software development system. The output of the job that created the tape, OSTAPE2, is included. It should be sufficient to determine how to unload the tape. The following paragraphs describe each file on the tape. Unless otherwise noted, each file is an unloaded partitioned data set. An alias filename is given whenever it may be referenced in two different ways.

File 1 - SUE.P7070.LINKLIB
(Alias: SUE.VERSION1.P7070.LINKLIB)

This library is used as the input to the PDP-11 tape generation process. It contains OS load modules with all external references resolved.

File 2 - SUE.GN.KERNEL.LINKLIB

The object deck output of compilations is run through the linkage editor and into this library - it contains load modules with unresolved external references. A subsequent link edit that resolves the external references uses this library as input and the File 1 library as output.

File 3 - TSØ231.SUE.GN.KERNEL.SOURCE

This file contains the source card images of most of the PDP-11 software. Most of the members contain SUE code, but some contain PAL-11 source. A copy of the PAL-11s cross assembler that we use can be obtained for $25.00 from

Mr. William F. Decker
University of Iowa
Iowa City, Iowa

File 4 - TSØ231.SUE.P7070.SOURCE
(Alias: SUE.VERSION1.P7070.SOURCE)

This file contains the source card images for the rest of our PDP-11 software.

File 5 - SUE.GN.KERNEL.DATA

File 6 - SUE.GN.KERNEL.PROGRAM

File 7 - SUE.P7070.DATA
(Alias: SUE.VERSION1.P7070.DATA)

File 8 - SUE.P7070.PROGRAM
(Alias: SUE.VERSION1.P7070.PROGRAM)

The members in these files are the output of the SCRUNCH preprocessor and the input to the SUE-11 compiler.

File 9 - SUE.DISTRIB.LOADMOD
(Alias: SUE.VERSION1.LOADMOD)

This file contains the object code of three programs - NIT, RELOCATE, and WLSALTER. NIT transforms the output of the PAL-11s cross assembler we use into an OS object deck so that further processing by the linkage editor can take place. NIT is not intended to be fully general or idiot proof, and will only work for small and simple programs (no external references) of the type that we have written. The catalogue procedure that invokes NIT is WLSPAL. The following control cards should be used:

```
//EXEC WLSPAL,FILE='<filename>',MEMBER=<name>,TSOID=TSØ231
//NIT.SYSIN DD *
<name>
```

RELOCATE has already been discussed.

The load module WLSALTER is used for maintaining source input as card images on a private disk pack. A listing of the user's source is always produced after execution of the ALTER program. This program is a modification of a MITRE utility program written in OS Assembler. Line numbers are added to the print file records when a change is made to the source file, but the OUTPUT subroutine's out file is never numbered. The ALTER program is referenced in the "Project 7070 versus IBM 370..." document (SUEAS) with further details found in the enclosed documentation entitled "Using the ALTER Program".

File 10 - SUE.WLS.SYSTEM.SOURCE
(Alias: SUE.VERSION1.SYSTEM.SOURCE)

This file contains the source of NIT, RELOCATE, and WLSALTER. NIT is written in PL/1 and can probably be compiled by the F or Optimizing compilers. RELOCATE and WLSALTER are written in OS Assembler. RELOCATE uses a few macros included on this tape.

File 11 - SUE.VERSION1.PROC

This file contains catalogued procedures. The procedures that begin with "SUE" are documented in "Project 7070 versus IBM 370..." - they are similar but not identical to the procedures distributed by the University of Toronto. The only other procedures are WLSPAL and WLSTAPE.

File 12 - WLS.MACLIB

OS Assembler macros used by RELOCATE.

File 13 - TSØ231.JCL
(Alias: SUE.VERSION1.JCL)

This file contains JCL and control cards to compile/assemble all of the PDP-11 source on the PDP-11 tape.

CONFIGURATION DEPENDENCIES

Our PDP-11 software has configuration dependencies imbedded in program constants and code. There are three types of dependency - main memory, secondary storage, and terminals. A constant in the context block KERNEL, MEM_SIZE, sets our memory size to 128K bytes. If you make this constant larger you must also increase the size of the Memory Block Table, and this change will require adjusting the locations of all the Kernel data structures that follow it. There should be no problems running in less than 128K bytes if restricted to two or three processes. Main Memory requirements and the impact on the number of processes that can be used have not yet been adequately determined.

The Kernel supports an RF11 for secondary storage. Only the PAL-11 routine DISKIO in the Kernel knows about the RF11. If you are willing to restrict your experimentation you can probably just turn off the disk I/O, removing the calls to DISKIO and the P's on the disk semaphore from SWAPIN and SWAPOUT. (See Dijkstra's "The" paper for a discussion of this topic.)

We currently support four TTY-like terminals. The UNIBUS addresses of the control registers for these terminals have been set so that they all begin at the same offset relative to a 256 byte segment. The included configuration chart gives a complete listing of our UNIBUS addresses. One terminal has the standard address for the system console. Changing the number of terminals requires, among other things, changing the constant USER_PROCESS#_MAX in the context block NOFORN, changing the code in STARTUP that initializes I/O vectors and wires down I/O segments, adding interrupt handlers to the GATE program in the Kernel, changing the constants in PROGRAM EXEC that define ASTE#'s for I/O and program code segments, and changing the code in EXEC that puts I/O segments into the hierarchy.

USING THE ALTER PROGRAM

The ALTER program uses two input files and three output files. The data to be updated is referenced as a sequential input file called IN. The second input file is used to contain the ALTER control cards, which specify additions or deletions of whole cards and the changes to a given card image. Alterations to be input file are made on the basis of their numeric sequence number as read in (not on the basis of any number which may appear on the card image).

When the ALTER program copies the user's input file, two output files are generated, one for the printer (PRINT) and the other for the user's updated version (OUT). The third sequential output file is used to list the control cards and any error messages.

The MFT and MVT versions of OS/370 will generate a //SYSIN DD * card automatically if they encounter unspecified data cards in the input stream. If SYSIN is present (including DD DUMMY and the implied DD * cases), then sequence numbers will be generated on the output records in columns 77-80; otherwise the records will be copied unmodified.

CONTROL CARDS

A control card is a card from the SYSIN data set having a number sign (#) in column one. There are three types of control cards (described below). All control cards have an integer immediately following the number sign. The control cards must be arranged so that these integers (alter numbers) are in strictly ascending order. A card from SYSIN that is not a control card is copied to OUT and PRINT files after the last mentioned alter number. Note that insertions may follow all types of control cards and that it is not possible to insert a card commencing with a number sign.

1. <u>Position control card</u>:
   #n

This card begins with a number sign, an integer, and two blanks; anything else on the card is ignored. It causes no change directly but is used to position the input for insertions.

2. <u>Deletion control card</u>:

   #n,m

This card begins with a number sign, an integer, a comma, another integer, and two blanks; anything else on the card is ignored. The second integer must be not less than the first. The corresponding cards from the IN file (n through m inclusive) are deleted (not copied to OUT and PRINT files). Replacements for the deleted cards can, of course, appear after the deletion control card.

3. <u>Alter control card</u>:
   #n A .pattern.replacement[.]

This card begins with a number sign, an integer, a blank, an "A", another blank, a control character, a pattern, another control character, a replacement string, and optionally a third control character. The control character may be any character (including blank). The pattern may be any string of one or more characters excluding the control character. The replacement may be any string; it is considered to end at the last non blank character on the card unless the character is a control character, in which case the replacement terminates at the character to the left of the last non blank character. Note that the replacement can have zero length or can itself contain instances of the control character. Note also that comments are not permitted on this control card.

The effect of the alter control card is to replace the first (from left to right) instance of the pattern with the replacement. The characters to the right of the pattern on the original card are moved left or right as necessary except that blanks are propagated left from column 77 (the sequence number field is not eligible for left shifts). Any characters shifted right from column 76 are lost. Only one replacement may be made on a card.

ERROR CONDITIONS

1. Alter number too large. (Return code 4.) This condition occurs if an alter number that is greater than the number of input records is encountered. The remaining input on SYSIN is ignored.

2. Illegal control card. (Return code 8.) This can mean an incorrect format or a card out of sequence; in either case it is ignored.

3. Match not found for alter. (Return code 16.) If no match is found then no change is made.

The return code from the ALTER program is the sum of the individual codes except that each is counted at most once. Thus, a return code of 20 would mean that SYSIN input was ignored, one or more alter matches failed, and there were no illegal control cards.

Example

```
11 EXEC  SUEAS,FILE='SAMPLE'
#2,2
    HELLO SUE
#38 A .DATA.
#44 A .DSN.DS.
#88,90
/*
```

The changes caused by the above control cards would be:

1. The second card image is replaced by the data line "HELLO SUE".

2. The first occurrence of "DATA" on the 38th card image is deleted.

3. The "DSN" is changed to "DS" on the 44th card image.

4. Card images 88 through 90 are deleted.

USING THE KERNEL EXERCISER

The basic functions of the Security Kernel as it is currently running on the PDP-11/45 may be accessed by way of the Kernel Exerciser. Outlined below are the procedures for calling up and running this program.

I. Steps to mounting the system tape and initiating execution:

1. Mount the system tape - PDPTAPE1 - on drive 0 and press LOAD. The ONLINE button should light.

2. On the system operator's console, the control knobs

for the Address Display Select should be set at CONSOLE PHYSICAL, and for the Data Display Select at DATA PATHS.

3. Boot load the system tape by entering 773136(octal) into the console switch register. Press in sequence HALT, LOAD ADDR, ENABLE, START.

4. The TTY Decwriter, and two TELTERM scopes should echo a carriage return (cr) signaling that the keyboards are unlocked. (Note that all communication is in TTY mode.)

II. Logging onto the system:

Every user logs onto the system by entering:

START /userid/ /projectid/ /classification/ /category/ (cr)

/userid/ - any decimal integer from 8 to 32766
/projectid/ - any decimal integer from 2 to 126
/classification/ - U (unclassified), S (secret), C (confidential), or T (top secret)
/category/ - any decimal integer from -32768 to 32767

The system response is: HELLO, THE KERNEL EXERCISER IS IN CONTROL

At this point, any of the Kernel commands, as described in Section IV, may be entered with the appropriate arguments.

III. Error Recovery

Should unrecognizable data be entered, the system response is:

ERROR, TRY AGAIN

Cancel a line by pressing the line feed key; correct a line by using the "@" key as a backspace key.

IV. The Kernel Commands

FUNCTION	ARGUMENTS	RETURN CODE
CREATE	/seg#/ /offset/ /class/ /cat/ /seg-type/ /size/	OK or ERROR
DELETE	/seg#/ /offset/	OK or ERROR
GIVE	/seg#/ /offset/ /mode/ /user/ /project/	OK or ERROR
RESCIND	/seg#/ /offset/ /user/ /project/	OK or ERROR
GETW	/seg#/ /offset/	/seg#/ or ERROR
GETR	/seg#/ /offset/	/seg#/ or ERROR
RELEASE	/seg#/	none
ENABLE	/seg#/ /reg#/	OK or ERROR
DISABLE	/reg#/	none
P	/seg#/	OK or ERROR
V	/seg#/	OK or ERROR
T	/seg#/	OK or ERROR
IPCSEND	/process#/ /message/	none
IPCRCV	none	(OK,/message/, /process#/
STOPP	none	STOPP ACCEPTED
READIR	/seg#/ /offset/	(OK,/class/, /cat/,/seg_type/, /size/) or ERROR

ARGUMENT VALUES

/class/:	1-4	1 - unclassified 2 - confidential 3 - secret 4 - top secret
/cat/:	0-32767	
/message/:	1-32767	
/mode/:	W - write access R - read access N - no access	
/offset/:	1-63	Root offsets - 1 - Process Directory Directory 2 - I/O Directory 3 - Code Directory

/process#/:	1 - executive 2 - TTY 3 - DECwriter 4 - Telterm #1 5 - Telterm #2	
/project/:	1-127	1 - System project 127 - All Projects
/reg#/:	0-15	0 - Stack reg 1 - I/O reg 2 - Code reg
/seg#/:	1-31	1 - Root seg 3 - Code seg 4 - Stack seg 5 - I/O seg
/seg-type/	0 - data 128 - directory	
/size/	1 - 256 bytes 4 - 1k bytes 16 - 4k bytes	Only size 4 (1K bytes) is implemented
/user/:	8-16383	16383 - All users

ERROR CODES
***** *****

-1:	OK	Operation performed
-2:	ERROR	Security or Implementation Violation
-3:	SEVERE ERROR	Parameter out of bounds or segment not in address space

PROJECT 7070 VERSUS IBM 370 OS/TSO AND THE PROJECT SUE SYSTEM LANGUAGE

The Project SUE System Language and compiler were chosen as tools to implement the security model on the PDP-11/45 because its structure allows the construction using structured programming (the language has no GOTO statement) of an operating system which, as far as a high level language and machine code is concerned, can be proved correct. This language supports Top-Down construction and testing of all modules within the limited purpose operating system being developed by Project 7070. The compiler and assembler

run on the IBM 370 and the conventions described here were invented to help the programmer to cope more easily with the intricacies of the program structure, compiler requirements and OS/370 and thus allow the programmer to concentrate on producing correct code.

All Project 7070 files are kept on-line and TSO is used to edit those files because it is faster than a batch editor and one avoids the pitfalls of keypunch errors, errors in JCL and long waits, among other things. Jobs can be submitted either through normal batch processing or through the Remote Job Entry (RJE) facility of TSO and output is received in the ordinary manner. If one wishes one can fetch output to an on-line data set and examine it at the terminal. These possibilities are limited only by budget and the programmer's imagination.

The Sue Program Structure

The SUE System Language is a block-structured language somewhat like ALGOL, but with COBOL-like structures known as compilation blocks. There exist in the language three types of these structures, Context Blocks, Data Blocks and Program Blocks. The Context Blocks contain declarations for global types, absolute locations for variables in virtual space and global Macro declarations. The Program Blocks contain executable code and all declarations of local variables and types if that is feasible.

Any Sue program or procedure must consist of a Data Block (which need not contain any statements) and a Program Block both of which have the same name in the Block Head (e.g., '<u>Data</u> NAME1;' are headings for the Data and Program Blocks respectively of a program called NAME1). If the program contains no internally called procedures, the Data and Program Blocks can be fed into the compiler one right after the other. If, on the other hand, internal procedures are declared, then the Program Block for any procedure must be preceded by its own data block and those of its enclosing procedures and must be followed by the Program Blocks of its enclosing procedures. The proper sequence for Data and Program Blocks is illustrated in Figure 1a. Note that the structure in Figure 1b is similar to nested Fortran DO loops and that the outermost Program Block may be omitted if the programmer does not wish it to be compiled. This compilation schedule is reponsible for the structure of the Project 7070 SUE files.

The Compilation Procedure

There are two steps in any compilation of the source code of the SUE System Language - 1) a preprocessing step called SCRUNCH which produces input to 2) the compile step. Because of this scheme every SUE file has 4 PDSs allocated to it for code - 1 PDS containing source code, 2 PDSs containing SCRUNCHED Code and 1 PDS containing link edited object code. Figures 2a and 2b summarize the functions and illustrate the naming conventions for these PDSs. The SOURCE PDS has members whose names are the procedure names which may be suffixed by either "D" or "P". For stand-alone Procedures, both the Data and Program Blocks are contained in one member with the same name as the procedure name. Procedures containing internal procedures have Data and Program blocks in separate members with names suffixed by "D" and "P" respectively. For example - a stand-alone procedure called NAME1 would have all its source code contained in a member of the SOURCE PDS called NAME1 - in other words it exists as one member in the source PDS; while a procedure named NAME2 containing internal procedures would have its Data Block contained in a SOURCE PDS member called NAME2D and its Program Block contained in a SOURCE PDS member called NAME2P - thus this program, unlike the stand-alone procedure, exists as 2 members in the SOURCE PDS rather than just 1. At the beginning and end of each member are control statements (beginning with "$$") which tell the Scrunch Preprocessor when an end of file condition has been reached (last statement of SOURCE PDS member) and where and under what name to place the output from the scrunch step (the first statement of the SOURCE PDS member). In Figures 3a and 3b we see that each member begins with the $$OUT_NAME statement. The parentheses immediately to the left of the equals sign contains the name of the particular Scrunch PDS (Data or Program) to which the output from the preprocessor will be read. All SOURCE member names ending with "D" will have "DATA" within the parentheses. Those ending with "P" will have "PROGRAM" within the parentheses. To the right of the equals sign is placed the name by which the member will be called. This name is, by convention, the Procedure or Program name unless that name is more than 8 letters or contains a break character ('_'). The last statement of each SOURCE PDS member is the $$EOF which indicates an end of records condition to the preprocessor.

```
CONTEXT     anyname
       .
       .
       source code     if needed
       .
       .
 _|_
DATA     NAME1
       .
       .
       source code    (if any)
       .
       .
 _|_
PROGRAM     NAME1
       .
       .
       source code
       .
       .
 _|_
```

Figure 1a. A SUE Program (Generalized) With No Internal Procedures

```
CONTEXT     anyname;  -if needed
_|_
DATA      NAME1 ;
DECLARE PROCEDURE   (NAME2);
_|_
DATA      NAME2 ;
DECLARE PROCEDURE   (NAME3,NAME4);
_|_
DATA      NAME3 ;
_ _
PROGRAM     (NAME3);
_|_
DATA      NAME4 ;
DECLARE PROCEDURE   (NAME5);
_|_
DATA      NAME5 ;
_|_
PROGRAM     (NAME5);
_|_
PROGRAM     (NAME4);
_|_
PROGRAM     (NAME2);
_|_
PROGRAM     (NAME1);  -can be omitted
_|_
```

Figure 1b. Schematic Diagram Showing Compilation Order and Scope of Various Blocks of a Procedure with Internally Declared Procedures

1)	SOURCE	- card images which are input to the SCRUNCH Preprocessor
2)	DATA	- output from SCRUNCH, input to compiler
3)	PROGRAM	- output from SCRUNCH, input to compiler
4)	LINKLIB	- Link edited object code, output of the compiler

Figure 2a. Names of SUE PDSs Which Contain Code and their Function

SUE. <filename>.<PDSname>

N.B The Source PDS has the programmer's TSO account number prefixed to the data set name.

<file name>, by convention, is the programmer's initials followed by a period and the filename (e.g., JAL.SYSPROC)

Figure 2b. Generalized Names for USE PDSs.

```
$$OUT_NAME(DATA)=NAME1
DATA  NAME1;

     .
     .
     source code
     .
     .

PROGRAM  NAME1;

     .
     .
     source code
     .
     .

$$EOF
```

Figure 3a. Generalized Diagram of Source Code for Member Name1 Showing Scrunch Control Cards. (This is a procedure without contained procedures. Resides in Source PDS as NAME1.)

```
$$OUT_NAME(DATA)=NAME2
DATA    NAME2;

       .
       .
       source code

$$EOF
```

Figure 3b.1 Data Block for a Program Called Program Name Which has Internal Procedures. (Resides in SOURCE PDS as NAME2D.)

```
$$OUT_NAME(PROGRAM)=NAME2
PROGRAM   NAME2;
          .
          .
          source code
          .
          .

 $$EOF
```

Figure 3b.2 Program Block for a Program Called Program Name Which Has Internal Procedures. (Resides in SOURCE PDS as NAME2P.)

The Scrunch Step

In order to call the Scrunch Preprocessor, the programmer codes a JCL EXEC control card calling the Procedure SUES specifying the file name (FILE='<file name>') and member name (MEMBER= <membername>). The procedure will invoke the Scrunch Program using the parameters given by the programmer to identify the Data Set Name and Member Name of the member of the SOURCE PDS to be scrunched. SUES must be invoked for every member to be scrunched and the procedure will overwrite any existing scrunched code in the member of the Scrunch Data Set if that member exists. If that member does not exist it will be created. Therefore, when changes are made to source code, that code can be scrunched without first having to delete the scrunched code associated with the original source code.

The Compilation

The compiler is invoked by one of 2 procedures - SUEC or SUECL. The first is intended to produce only a listing with error messages, the data set containing object code being deleted by OS. The second procedure passes the object code to the Linkage Editor, which it invokes.

When calling SUEC the programmer must specify the filename (FILE='<filename>') and include control records after the EXEC card which tell the compiler which members of what Scrunch PDS to compile and in what order these members are to be compiled. Figures 4a and 4b give the general form and an example.

Link Editing

SUECL is called in much the same way and with the same control record scheme. The only difference is in the addition of a specification of a member name (MEMBER=<membername>). This will be substituted in the JCL for this procedure to name a member in the LINKLIB PDS to which the link-edited object code will be written. The Linkage Editor is invoked in this procedure with the NCAL option specified. This allows the storage of object modules with unresolved external references until the segments of code to which they refer have been coded, compiled and separately link-edited. After the programmer has coded all procedures for a particular system of procedures (i.e., after all internal procedures have been coded, compiled and object modules placed in the LINKLIB PDS), SUEL is called to invoke the Linkage editor, the NCAL option, in order to completely link edit all object modules of a system of procedures and store the resultant load module in the Linklib PDS. The programmer is required to specify in the EXEC Statement a value for a member name in the LINKLIB PDS (MEMBER= membername) to which the load module is to be written and s/he will be required to use control statements of the form - INCLUDE SYSLIB(<member name>) - which indicates to the Linkage Editor which members (<membername>) of the Linklib PDS are to be processed. There will be as many of these statements as there are members to be link edited. The process will bomb if there are any procedure calls for which no procedure has been coded, compiled and stored in the Linklib PDS.

USING TSO

In order for the Programmer to use TSO each file is allocated 2 additional PDSs: 1) a CNTL PDS with a data set name of

```
<scrunch PDSname><member name><control toggles>

     <scrunchPDSname> is either PROGRAM or DATA

     <member name>  is the member of the scrunch PDS to
                    be compiled

     <control toggle> controls the emission of information
                      to the Sysprint data set
```

Figure 4a. General Form for Compiler Control Records

```
//      EXEC SUEC, FILE='<file name>'

DATA NAME1        ¬L       (no listing of this block)

DATA NAME2         D       (list symbol and type tables)

DATA NAME3

PROGRAN NAME3

DATA NAME 4

PROGRAM NAME4

PROGRAM NAME2      DE      (list symbol and type tables,
                            emitted code)

/*
```

Figure 4b. Example of Invocation of Compiler with Control Records

<tsoid>. CNTL and 2) A CLIST PDS with a data set name of <tsoid>. X.CLIST. The first is storage for JCL card images and other control cards for submission to batch processing through the Remote Job Entry (RJE) facility of TSO (see caveat #3 before using RJE). The second is storage for any command lists which a programmer may want to create. One useful command list passes values for a file name and a member name to a generalized call to the editor for a member of a PDS and executes the call. Other uses are left to the programmer's imagination.

Project 7070 uses a catalogued procedure to allocate PDSs for files. The programmer calls the catalogued Procedure SUEALLOC and specifies his TSO account number (TSOID=<tsoid>) and the file name (FILE=<filename>).

USING MITRE BATCH FACILITIES

All procedures mentioned here can be called in the batch job stream. Members can be created by IEBUPDTE; they can be updated by using the procedure SUEAS which calls the ALTER program and uses the same utility control statements (see the Facility Manual). In addition this procedure rewrites the Scrunched data set. There is a card to printer procedure called SUESC which takes card input and produces a compiler diagnostic. With TSO, however, these two procedures are not necessary and all the others may be called through JCL which the programmer has stored in dynamically created CNTL data sets. Of course, in order to use TSO the programmer has to have some way of identifying himself to the system as a legitimate user and this is done through the TSO account number.

GETTING READY FOR TSO

In order to obtain a TSO account number (which is also the "TSOID") the programmer can call User Assistance at X2525, and after giving Lee Gera the pertinent information, he will assign the programmer a number of the form TSØXXX and an eight character Password. After Lee places the programmer's account on the system the programmer may proceed to do his thing. However, to do it on TSO s/he must first read the TSO User's guide (GC28-6697) which will introduce her/him to TSO and have ready the TSO Command Language Reference which will serve as a reference guide to the Function, Syntax and Operands of TSO commands. In addition, the programmer will have to have a MITRE Computer Facilities Manual in order to see the changes and extensions that MITRE has made to the original IBM Product in order to make it more compatible with humans. These changes and extensions are listed and explained in Chapter 7. All these and any other manuals may be obtained from Stella Theokas in the keypunch area of the computer facility.

Caveats

(1) It may not be obvious from the following attachments that a SUE file resides on 2 disk packs. The SOURCE, CNTL and CLIST PDS-reside on the public pack and the DATA, PROGRAM and LINKLIB PDSs reside on a private pack (serial number DP5006). With the present LOGON procedure available to Project 7070 (the default procedure), only those PDSs residing on the public packs can be accessed through TSO. However, others may be assessed by any Job entered through RJE.

(2) All control cards for the compiler and Linkage Editor begin in column 1.

(3) One slight failing of RJE is that there is no way for the operator to know that a private storage medium is required unless the JCL for the Job tells him. This is done at MITRE through a HASP SETUP card image. This is described in the Facilities Manual on Page 6.2. The user should request Volume DP5006 - a disk pack.

Summary with Examples of SUE Procedure Calls

1) SUEALLOC: Allocates the 6 PDSs required for working with the SUE language. SOURCE, CNTL and CLIST are allocated on the Public Packs while DATA, PROGRAM and LINKLIB are allocated to the Project 7070 private Pack. File name must be specified.

e.g., //anyname EXEC SUEALLOC,FILE='JAL.SYSPROC',TSOID=TSØ999

2) SUES: Scrunches source code and places the result on either the Data or Program PDS according to what control records appear with the source code (see Figures 3a and 3b). The programmer's TSOID, file name and member name of the Source PDS member to be processed, must be specified.

e.g., //anyname EXEC SUES,FILE='JAL.SYSPROC',MEMBER=DELETEP,
TSOID-TSØ999

3) SUEC: Compiles scrunched code, produces listings and diagnostics and scratches files containing object code. Programmer's file name must be specified and control records must be placed after the EXEC CARD (see the section on compilation and Figure 5b).

```
e.g.,   //anyname  EXEC SUEC,FILE='JAL.SYSPROC'
                *
                *
compiler control records
                *
                *
     /*
```

N.B. the member name specified is the member in the LINKLIB PDS to which the object code will be written.

5) SUEL: Link edits files specified by control records (see section on Link Editing) and terminates at discovery of unresolved external reference or at successful completion. When successful, places fully link edited code in LINKLIB PDS under a specified member name. File name and member name must be given.

```
e.g.,   //anyname  EXEC SUEL, FILE='JAL.SYSPROC',MEMBER=DELETE
                *
                *
       Linkage Editor
       Control Records
                *
                *
       /*
```

6) SUEAS: Batch editor for Project 7070. Calls Alter and Scrunch Programs. Filename and member from SOURCE PDS must be specified. Control cards and change cards are as for the ALTER Program. See the Facility Manual Page 5.80 for more information.

```
e.g.,   //anyname  EXEC SUEAS, FILE='JAL.SYSPROC',MEMBER=DELETEP
                *
                *
       control and change cards for ALTER
                *
                *
       /*
```

7) SUESC: Executes scrunch and compile on card input. Output is to printer for listing and diagnostics. All files are temporary.

```
e.g.,   //anyname  EXEC SUESC
                *
                *
       card input
                *
                *
       /*
```

8) SUESCRTH: Deletes all PDSs for a particular file name. TSOID and file name must be specified.

e.g., //anyname EXEC SUESCRTH,TSOID=TSØ999,FILE='JAL.SYSPROC'

Examples of Program Execution and Development Under TSO

A guide and scenarios are given below to illustrate the steps a SUE programmer must use to allocate files for her/himself, create and edit PDS members and submit jobs.

Allocation of Files

The allocation of files can be done either through TSO or through the batch. It seems frivolous to spend money on the submission of such a short job through RJE so batch processing is recommended. The cards needed are a job card and the execute card for SUEALLOC (see example in the previous section). Don't forget to put the Serial # DP5006 on the back of the green request card under "Disk Packs Required".

Creating Members in Your Newly Allocated Disk Area

There are two ways of doing this - IEBUPDTE with cards or through TSO. IEBUPDTE JCL is first given. For a more comprehensive look at this utility the programmer should consult the OS Utilities Manual (GC28-6586).

```
//jobcard (if necessary)
// EXEC PGM=IEBUPDTE, PARAM-NEW
//SYSPRINT DD SYSOUT=A
//SYSUT1   DD DSN=<tsoid>.<filename>. SOURCE,DISP=SHR
//SYSUT2   DD DSN=<tosid>.<filename>. SOURCE,DISP=SHR
//SYSIN    DD   *
./ ADD LIST=ALL,NAME= membername1
./ ADD LIST=ALL,NAME= membername2
          (card images)

                *
                *
                *

/*
```

N.B. <membername1> and <membername2> are the names the programmer has chosen according to the conventions stated previously for source PDS members. There may be any number of ADD statements-one for each member to be added.

To do the above through TSO one would make the following call to the editor:

```
edit '<tsoid>.SUE.<filename>.SOURCE(<membername>)'
DATA NEW NONUM
```

Since the member is new the user is placed into input mode immediately and is ready to write whatever input s/he wants. One must do this for each member to be placed on the PDS. Typing out such a long command can become tedious when one does it over and over again - as one sometimes must for a large series of editing changes to many members of many data sets - therefore, the CLIST facility is used. The following command is used to create the member whose membername is 'e' in the CLIST PDS named X:

```
edit x(e)  clist new
```

When the terminal is in input mode the following entries are made:

```
proc 2 fname mem

edit'<tsoid>.sue.&fname..source(&mem.)'data nonum
```

N.B. This is for editing an already existing member. If one wanted to create a new member one would place the keyword "NEW" after the attribute parameter. A carriage return brings the terminal to edit mode and the command

```
se
```

saves and ends the edit session. To learn more about the PROC statement in TSO please consult the TSO Command Language Reference. When one wants to call the editor one types in the exec command specifying the file name and member name like so:

```
exec x(e) '<filename><membername>'
```

The terminal will be returned to you in Edit mode for the member specified, or input mode if NEW is specified in the CLIST.

In order to run jobs through RJE one must have JCL card images on disk and accessible to the terminal. The CNTL PDS is the data set which will contain the JCL necessary. In order to place the JCL into members of the PDS one uses the EDIT command to create the new member:

```
edit cntl(<membername>) cntl new nonum
```

Now the terminal is in input mode and one can enter the JCL and, when necessary, any other information such as compiler or linkage editor control records and setup records. Here is a sample scenario where members of the CNTL PDS are created for scrunching a program called WRITES (which contains internal procedures) and compiling it with already scrunched NOFORN (a context block), WHYNOT (another context block), SEARCH (a stand along procedure), APPENDI (which has internal procedures) and APPENDS (another stand alone procedure), all of which are in the SUE file 'JAL.SYSPROC'. <CR> means that the carriage return button is pushed after a line is entered. The command SE saves the input and/or changes and exits from the command EDIT.

1) Make a Job Card

```
Command:  edit 'tsØ999.cntl(jobcard)' cntl new nonum

Input:    //TSØ999A  JOB  (7070,D73, DESK),'LARKINS JA',CLASS=D
          // REGION=256K,TIME=(,20),NOTIFY=TSØ999

          <CR>

Command:  se
```

2) Scrunch both the DATA and PROGRAM blocks of WRITES

```
Command:  edit 'tsØ999.cntl(SWRITES)' cntl new nonum

Input:    //SCRUNCHD   EXEC   SUES,FILE='JAL.SYSPROC',
                              MEMBER-WRITESD,TSOID=TSØ999

          //SCRNCHP    EXEC   SUES,FILE='JAL.SYSPROC',
                              MEMBER-WRITSP,TSOID=TSØ999

          <CR>

Command:  se
```

3) Compile the program and produce only diagnostics and listing.

```
Command:  edit 'tsØ999.cntl(cmpwrite)' cntl new nonum

Input:    //CMPWRITE  EXEC    SUEC,FILE='JAL.SYSPROC'
          DATA NOFORN L       (no listing)
          DATA WHYNOT
          DATA WRITES D          (sysbol and type table listed)
          DATA SEARCH
```

```
DATA APPENDI
DATA APPENDS
PROGRAM APPENDI
PROGRAM WRITES   DE      (Emitted Code, Symbol table
                         and type table listed)

<CR>
```

```
Command:  se
```

4) Submit the job through RJE

```
Command:  sub (cntl(jobcard) cntl(swrites)cntl(cmpwrite))
```

A HASP job number will be returned by the system and should be copied down for use by the STATUS command and other commands which are outlined in the Facility Manual.

If the programmer wanted to link edit the object code and store it the following step would replace step 3.

```
Command:  edit 'tsØ999.cntl(cmpwrite)' cntl new nonum

Input:    //CMPWRITE EXEC SUECL,FILE='JAL.SYSPROC',MEMBER=WRITES
          DATA NOFORN        ¬L
          DATA WHYNOT        ¬L
          DATA WRITES         D
          DATA SEARCH        ¬L
          DATA APPENDI       ¬L
          DATA APPENDS       ¬L
          PROGRAM APPENDI    ¬L
          PROGRAM WRITES      E
```

The link edited object code will be stored in the member WRITES of the LINKLIB PDS. The submit command would look the same.

If the programmer had finally coded his whole system of procedures and placed each of their object codes in the LINKLIB PDS and wanted to obtain fully link edited object codes s/he would code the following JCL.

We assume that the members WRITEN, SPVRGATE, CHANGEB, IYPYE, READS and READN are members of LINKLIB.

```
Command:    edit 'ts∅999.cntl(fileproc)' cntl new nonum

Input:      //LINKED  EXEC SUEL,FILE='JAL.SYSPROC',MEMBER=FILEPROC
             INCLUDE  SYSLIB(SPVRGATE)
             INCLUDE  SYSLIB(WRITES)
             INCLUDE  SYSLIB(WRITEN)
             INCLUDE  SYSLIB(READS)
             INCLUDE  SYSLIB(REDN)
             INCLUDE  SYSLIB(ITYPE)
             INCLUDE  SYSLIB(CHANGEB)

             /*

             <CR>
Command:     se
```

Now to submit the job the programmer will issue the following command:

```
SUBMIT (CNTL(JOBCARD) CNTL(FILEPROC))
```

If the programmer wants to delete any members from the PDS on the PUBLIC PACK (SOURCE,CNTL,CLIST) s/he must use the delete command. The following is a command that deletes CNTL member SWRITES.

```
delete cntl(swrites)
```

The 3270 Display Terminal

The FSE subcommand of EDIT utilizes the 3270 display terminal to its fullest to ease the updating of card images. It can be used to edit the CNTL and SOURCE PDS members. This program enables the user to directly change her/his code on the screen without the mediation of the insert, delete, and change subcommands. For more information see pp. 7-5.2 ff of the facility manual.

The next section contains listings of the JCL for SUE Procedures.

```
//SUEALLOC PROC
//IEFBR14 EXEC PGM=IEFBR14,ACCT=COST
//SOURCE    DD  UNIT=PUBLIC,DISP=(NEW,CATLG),
//              DCB=(DSORG=PO,RECFM=FB,BLKSIZE=3120,LRECL=80),
//              SPACE=(TRK,(15,10,10)),
//              DSN=&TSOID..SUE.&FILE..SOURCE
//DATA      DD  UNIT=PACK,VOL=(PRIVATE,SER=DP5006),DISP=(NEW,KEEP),
//              DCB=(BLKSIZE=2048,RECFM=F),SPACE=(2048,(100,100,100)),
//              DSN=SUE.&FILE..DATA
//PROGRAM   DD  UNIT=PACK,VOL=(PRIVATE,SER=DP5006),DISP=(NEW,KEEP),
//              DCB=(BLKSIZE=2048,RECFM=F),SPACE=(2048,(50,50,50)),
//              DSN=SUE.&FILE..PROGRAM
//LINKLIB   DD  UNIT=PACK,VOL=(PRIVATE,SER=DP5006),DISP=(NEW,KEEP),
//              DCB=(DSORG=PO,RECFM=U,BLKSIZE=7294),
//              SPACE=(TRK,(20,10,10)),
//              DSN=SUE.&FILE..LINKLIB
//CNTL      DD  UNIT=PUBLIC,DISP=(NEW,CATLG),
//              DCB=(DSORG=PO,RECFM=FB,BLKSIZE=2000,LRECL=80),
//              SPACE=(TRK,(5,5,10)),
//              DSN=&TSOID..CNTL
//X         DD  UNIT=PUBLIC,DISP=(NEW,CATLG),
//              DCB=(DSORG=PO,RECFM=VB,BLKSIZE=1680,LRECL=255),
//              SPACE=(TRK,(1,1,2)),
//              DSN=&TSOID..X.CLIST
```

```
//SUEAS    PROC SCANNER=SCRUNCH,JOBNAME=SUEGROUP,SCROUT=DUMMY
//ALTER    EXEC PGM=WLSALTER,ACCT=COST
//STEPLIB   DD  DSN=SUE.DISTRIB.LOADMOD,DISP=SHR
//SYSPRINT DD   SYSOUT=A
//SYSUDUMP DD   SYSOUT=A
//PRINT     DD  SYSOUT=A
//IN        DD  UNIT=PACK,VOL=(PRIVATE,SER=DP5006),DISP=SHR,
//              DSN=SUE.&FILE..SOURCE(&MEMBER)
//OUT       DD  UNIT=PACK,VOL=(PRIVATE,SER=DP5006),DISP=(OLD,PASS),
//              DSN=SUE.&FILE..SOURCE(&MEMBER)
//*
//*
//SCRUNCH EXEC PGM=XMON,REGION=100K,
//              PARM='FREE=30000,JOBNAME=&JOBNAME',ACCT=COST
//STEPLIB   DD  DSN=SUE.DISTRIB.LOADMOD,DISP=SHR
//PROGRAM   DD   DSN=SUE.DISTRIB.&SCANNER,DISP=SHR
//SYSPRINT DD   &SCROUT
//SYSPUNCH DD   DUMMY
//FILE1     DD  VOL=(PRIVATE,SER=DP5006),DSN=SUE.&FILE..PROGRAM,
//              UNIT=PACK,DISP=OLD,
//FILE2     DD  VOL=(PRIVATE,SER=DP5006),DSN=SUE.&FILE..DATA,
//              UNIT=PACK,DISP=OLD,
//FILE6     DD  VOL=(PRIVATE,SER=DP5006),DSN=SUE.&FILE..PROGRAM,
//              UNIT=PACK,DISP=OLD,
//FILE7     DD  VOL=(PRIVATE,SER=DP5006),DSN=SUE.&FILE..DATA,
//              UNIT=PACK,DISP=OLD,
//OUTPUT3   DD  DSN=&&MODNAME,UNIT=SYSDA,DISP=(MOD,PASS),SPACE=(TRK,1),
//              DCB=BLKSIZE=80
//SYSIN     DD  DSN=*.ALTER.OUT,DISP=SHR
```

```
//SUESC    PROC SCANNER=SCRUNCH,SCROUT=DUMMY,VERSION=SUE11,
//              FREE=12000,JOBNAME=SUEGROUP,CNTRLD=60000
//SCRUNCH EXEC PGM=XMON,REGION=100K,
//              PARM='FREE=30000,JOBNAME=&JOBNAME',ACCT=COST
//STEPLIB   DD  DISP=SHR,DSN=SUE.DISTRIB.LOADMOD
//PROGRAM   DD  DISP=SHR,DSN=SUE.DISTRIB.&SCANNER
//SYSPRINT DD   &SCROUT
//SYSPUNCH DD   DUMMY
//FILE1     DD  DSN=&&TOKENS1,UNIT=SYSDA,DISP=(MOD,PASS),
//              SPACE=(TRK,(10,5,17)),
//              DCB=(DSORG=PO,RECFM=F,BLKSIZE=2048)
//FILE2     DD  DSN=&&TOKENS2,UNIT=SYSDA,DISP=(MOD,PASS),
//              SPACE=(TRK,(10,5,17)),
//              DCB=(DSORG=PO,RECFM=F,BLKSIZE=2048)
//FILE6     DD  DUMMY
//FILE7     DD  DUMMY
//OUTPUT3   DD  DSN=&&MODNAME,UNIT=SYSDA,DISP=(MOD,PASS),SPACE=(TRK,1),
//              DCB=BLKSIZE=80
//*
//*
//SUE      EXEC PGM=XMON,REGION=310K,COND=(C,LT,SCRUNCH),ACCT=COST,
//              PARM='FREE=&FREE,CONTROLD=&CNTRLD,JOBNAME=&JOBNAME'
//STEPLIB   DD  DISP=SHR,DSN=SUE.DISTRIB.LOADMOD
//PROGRAM   DD  DISP=SHR,DSN=SUE.DISTRIB.&VERSION
//SYSPRINT DD   SYSOUT=A,DCB=(RECFM=FBA,LRECL=133,BLKSIZE=532,BUFNO=2)
//FILE1     DD  DSN=*.SCRUNCH.FILE1,DISP=(OLD,PASS)
//FILE2     DD  DSN=*.SCRUNCH.FILE2,DISP=(OLD,PASS)
//OUTPUT3   DD  DSN=&&SYSLIN,UNIT=SYSDA,DISP=(MOD,PASS),
//              SPACE=(TRK,(20,5)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
//SYSIN     DD  DSN=&&MODNAME,DISP=(OLD,DELETE)
```

```
//SUES     PROC SCANNER=SCRUNCH,JOBNAME=SUEGROUP
//SCRUNCH EXEC PGM=XMON,REGION=100K,ACCT=COST,
//             PARM='FREE=30000,JOBNAME=&JOBNAME'
//STEPLIB   DD DSN=SUE.DISTRIB.LOADMOD,DISP=SHR
//PROGRAM   DD  DSN=SUE.DISTRIB.&SCANNER,DISP=SHR
//SYSPRINT DD  DUMMY
//SYSPUNCH DD  DUMMY
//FILE1     DD VOL=(PRIVATE,SER=DP5006),DSN=SUE.&FILE..PROGRAM,
//             UNIT=PACK,DISP=OLD
//FILE2     DD VOL=(PRIVATE,SER=DP5006),DSN=SUE.&FILE..DATA,
//             UNIT=PACK,DISP=OLD
//FILE6     DD VOL=(PRIVATE,SER=DP5006),DSN=SUE.&FILE..PROGRAM,
//             UNIT=PACK,DISP=OLD
//FILE7     DD VOL=(PRIVATE,SER=DP5006),DSN=SUE.&FILE..DATA,
//             UNIT=PACK,DISP=OLD
//OUTPUT3   DD DSN=&&MODNAME,UNIT=SYSDA,DISP=(MOD,PASS),SPACE=(TRK,1),
//             DCB=BLKSIZE=80
//SYSIN     DD DSN=&TSOID..SUE.&FILE..SOURCE(&MEMBER),DISP=SHR
```

```
//SUEC      PROC VERSION=SUE11,FREE=12000,
//               JOBNAME=SUEGROUP,CNTRLD=60000,SYS=P7070
//SUE       EXEC PGM=XMON,REGION=310K,ACCT=COST,
//               PARM='FREE=&FREE,CONTROLD=&CNTRLD,JOBNAME=&JOBNAME'
//STEPLIB   DD   DSN=SUE.DISTRIB.LOADMOD,DISP=SHR
//PROGRAM   DD   DSN=SUE.DISTRIB.&VERSION,DISP=SHR
//SYSPRINT  DD   SYSOUT=A,DCB=(RECFM=FBA,LRECL=133,BLKSIZE=532,BUFNO=2)
//FILE1     DD   VOL=(PRIVATE,SER=DP5006),DSN=SUE.&FILE..PROGRAM,
//               UNIT=PACK,DISP=SHR
//          DD   VOL=(PRIVATE,SER=DP5006),DSN=SUE.&SYS..PROGRAM,
//               UNIT=PACK,DISP=SHR
//FILE2     DD   VOL=(PRIVATE,SER=DP5006),DSN=SUE.&FILE..DATA,
//               UNIT=PACK,DISP=SHR
//          DD   VOL=(PRIVATE,SER=DP5006),DSN=SUE.&SYS..DATA,
//               UNIT=PACK,DISP=SHR
//OUTPUT3   DD   DSN=&&SYSLIN,DISP=(MOD,PASS),UNIT=SYSDA,
//               SPACE=(TRK,(20,5)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
```

```
//SUECL    PROC VERSION=SUE11,CNTRLD=60000,
//              FREE=12000,JOBNAME=SUEGROUP,SYS=P7070
//SUE      EXEC PGM=XMON,REGION=310K,ACCT=COST,
//              PARM='FREE=&FREE,CONTROLD=&CNTRLD,JOBNAME=&JOBNAME'
//STEPLIB   DD  DSN=SUE.DISTRIB.LOADMOD,DISP=SHR
//PROGRAM   DD  DSN=SUE.DISTRIB.&VERSION,DISP=SHR
//SYSPRINT  DD  SYSOUT=A,DCB=(RECFM=FBA,LRECL=133,BLKSIZE=532,BUFNO=2)
//FILE1     DD  VOL=(PRIVATE,SER=DP5006),DSN=SUE.&FILE..PROGRAM,
//              UNIT=PACK,DISP=SHR
//          DD  VOL=(PRIVATE,SER=DP5006),DSN=SUE.&SYS..PROGRAM,
//              UNIT=PACK,DISP=SHR
//FILE2     DD  VOL=(PRIVATE,SER=DP5006),DSN=SUE.&FILE..DATA,
//              UNIT=PACK,DISP=SHR
//          DD  VOL=(PRIVATE,SER=DP5006),DSN=SUE.&SYS..DATA,
//              UNIT=PACK,DISP=SHR
//OUTPUT3   DD  DSN=&&SYSLIN,DISP=(MOD,PASS),UNIT=SYSDA,
//              SPACE=(TRK,(20,5)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
//*
//*
//LKED     EXEC PGM=IEWL,REGION=96K,COND=(0,LT,SUE),
//              PARM=(XREF,LIST,NCAL),ACCT=COST
//STEPLIB   DD  DSN=SUE.DISTRIB.LOADMOD,DISP=SHR
//SYSPRINT  DD  SYSOUT=A
//SYSLIN    DD  DSN=*.SUE.OUTPUT3,DISP=(OLD,DELETE)
//          DD  DDNAME=SYSIN
//SYSUT1    DD  DSN=&&SYSUT1,UNIT=SYSDA,SPACE=(TRK,(50,5))
//SYSLMOD   DD  UNIT=PACK,VOL=(PRIVATE,SER=DP5006),DISP=OLD,
//              DSN=SUE.&FILE..LINKLIB(&MEMBER)
//SYSIN     DD  DUMMY
```

```
//SUEL     PROC SYS=P7070
//LKED     EXEC PGM=IEWL,PARM='XREF,LIST',ACCT=COST
//SYSPRINT DD   SYSOUT=A
//SYSLIN   DD   DDNAME=SYSIN
//SYSLMOD  DD   UNIT=PACK,VOL=(PRIVATE,SER=DP5006),DISP=OLD,
//              DSN=SUE.&SYS..LINKLIB(&MEMBER)
//SYSUT1   DD   UNIT=(DISK,SEP=SYSLMOD),SPACE=(1024,(200,20))
//SYSLIB   DD   VOL=(PRIVATE,SER=DP5006),DSN=SUE.&FILE..LINKLIB,
//              UNIT=PACK,DISP=SHR
```

```
//SUESCRTH PROC
//IEFBR14  EXEC PGM=IEFBR14,ACCT=COST
//CNTL      DD  DISP=(OLD,DELETE),DSN=&TSOID..CNTL
//X         DD  DISP=(CLD,DELETE),DSN=&TSOID..X.CLIST
//SOURCE    DD  DISP=(OLD,DELETE),DSN=&TSOID..SUE.&FILE..SOURCE
//PROGRAM   DD  DISP=(OLD,DELETE),UNIT=PACK,VOL=(PRIVATE,SER=DP5006),
//              DSN=SUE.&FILE..PROGRAM
//DATA      DD  DISP=(OLD,DELETE),UNIT=PACK,VOL=(PRIVATE,SER=DP5006),
//              DSN=SUE.&FILE..DATA
//LINKLIB   DD  DISP=(OLD,DELETE),UNIT=PACK,VOL=(PRIVATE,SER=DP5006),
//              DSN=SUE.&FILE..LINKLIB
```

PDP-11/45 Interrupt Vectors and Control Registers - 2 June 1975

Definition	Octal	Hex
Interrupt Vectors		
Reserved	0	0
Time Out, Bus Error, Odd Address, Stack Violation	4	4
Reserved Instruction	10	8
Debugging	14	E
IOT	20	10
Power Fail	24	14
EMT	30	18
TRAP	34	1C
Dec Writer In (BR4)	60	30
DEC Writer Out (BR4)	64	34
Programmable Real-Time Clock (KW11-P, BR6)	104	44
General Purpose DMA Interface (DR11-B, BR5)	124	54
Line Printer Control (LS11, BR4)	200	80
Disk Control (RF11, BR5)	204	84
DEC Tape Control (TC11, BR6)	214	8C
Magnetic Tape Control (TM11, BR6)	224	94
Card Reader Control (CR11, BR6)	230	98
Programming Interrupt Request	240	A0
Memory Management Unit Traps and Violations	250	A8
DIVA Disk (BR5)	254	AC
Start of Floating Vectors (BR5 unless otherwise noted, each device has a pair of vectors - in and out)		
Telterm #1 Control (BR4)	300	C0
Telterm #2 Control (BR4) (BR-90 Lab)	310	C8
Asynchronous Line Interfaces (DC11-DA)		
#1	320	D0
#2	330	D8

Definition	Octal	Hex
#3	340	E0
#4	350	E8
Synchronous Interface (DP11-DA) BR90	360	F0
TTY	370	F8
End of Floating Vectors	377	FF
<u>Control Registers</u>		
TTY (4 registers)	760160	3E070
DIVA Disk (8 words)	764000	3E800
Programmable Read-Time Clock (KW11-P) Secure Mode		
Count and Status	770540	3F160
Count Set Buffer	770542	3F162
Counter	770544	3F164
Supervisor Segmentation Registers		
I Space Descriptor Registers (0-7)	772200	3F480
D Space Descriptor Registers (0-7)	772220	3F490
I Space Address Registers (0-7)	772240	3F4A0
D Space Address Registers (0-7)	772260	3F4B0
Kernel Segmentation Registers		
I Space Descriptor Registers (0-7)	772300	3F4C0
D Space Descriptor Registers (0-7)	772320	3F4D0
I Space Address Registers (0-7)	772340	3F4E0
D Space Address Registers (0-7)	772360	3F4F0
General Purpose DMA Interface (DR11-B)		
Word Count	772410	3F508
Bus Address	772412	3F50A
Status and Command	772414	3F50C
Data Buffer	772416	3F50E
11/45 SSR3	772516	3F54E

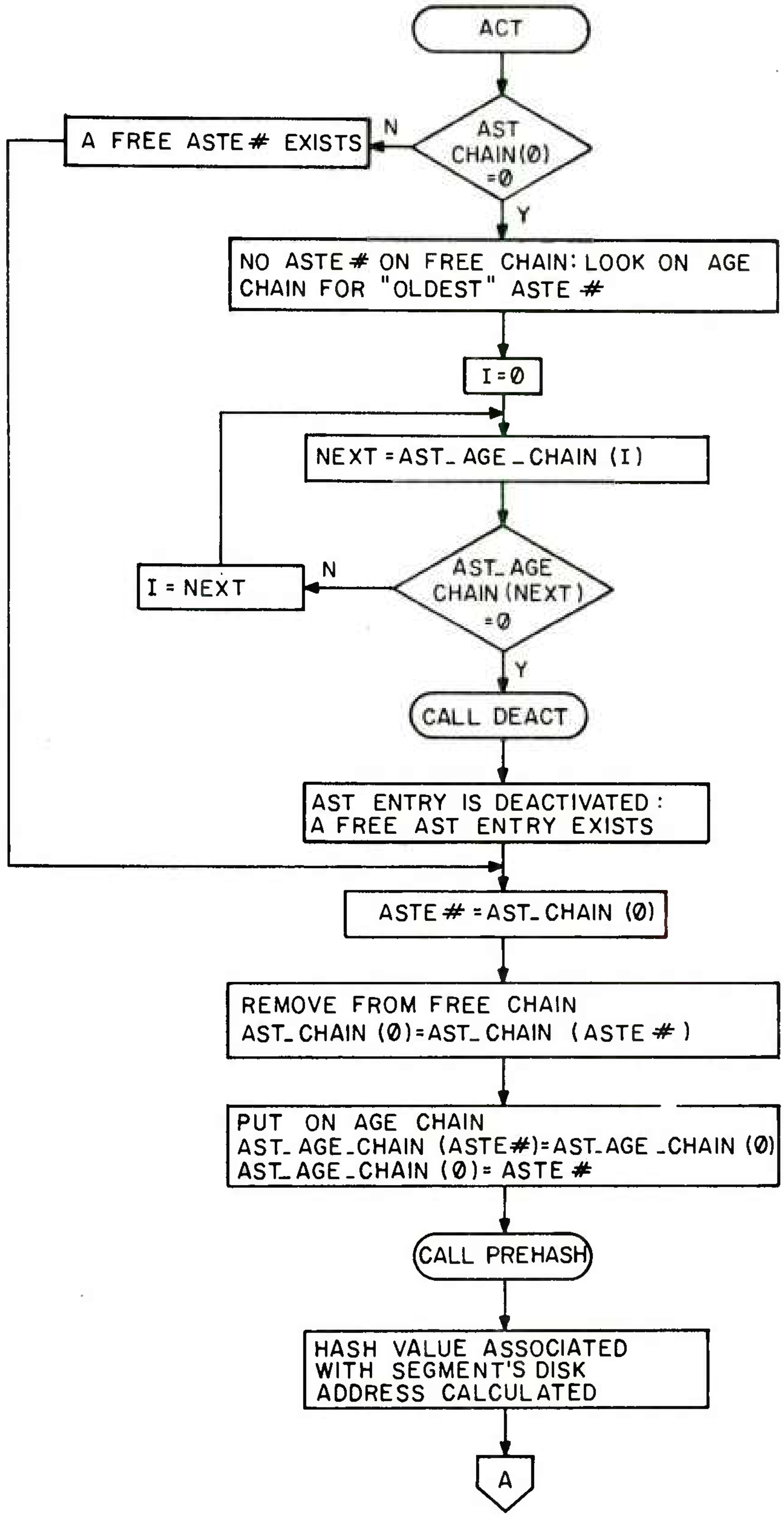
Definition	Octal	Hex
Magnetic Tape (TM11)		
Status	772520	3F550
Control	772522	3F552
Byte Counter	772524	3F554
Memory Address	772526	3F556
Data Buffer	772530	3F558
Read Lines	772532	3F55A
Programmable Real-Time Clock (KW11-P) Normal Mode		
Count and Status	772540	3F560
Count Set Buffer	772542	3F562
Counter	772544	3F564
Bootstrap Loader		
Disk (RF11)	773100	3F640
DEC Tape (TC11)	773120	3F650
Magnetic Tape (TM11)	773126	3F65E
Asynchronous Line Adapters (DC11-DA, 4 registers each)		
#1	774000	3F800
#2	774010	3F808
#3	774020	3F810
#4	774030	3F818
Synchronous Interface (DP11-DA, 6 registers/8 bytes) BR90	774770	3F9F8
Line Printer (LS11) Secure Mode		
Status	775564	3FB74
Data Buffer	775566	3FB76
Telterm #1 (4 registers)	776160	3FC70
Telterm #2 (4 registers) (BR-90 Lab)	776560	3FD70

Definition	Octal	Hex
Card Reader (CR11)		
Status	777160	3FE70
Data Buffer	777162	3FE72
Data Buffer	777164	3FE74
DEC Tape (TC11)		
Control and Status	777340	3FEE0
Command	777342	3FEE2
Word Count	777344	3FEE4
Bus Address	777346	3FEE6
Data Register	777350	3FEE8
Disk (RF11)		
Control Status	777460	3FF30
Word Count	777462	3FF32
Current Memory Address	777464	3FF34
Disk Address	777466	3FF36
Disk Error Status	777470	3FF38
Disk Data Buffer	777472	3FF3A
Maintenance	777474	3FF3C
Address of Disk Segment	777476	3FF3E
Line Printer (LS11) Nornal Mode		
Status	777514	3FF4C
Data Buffer	777516	3FF4E
DEC Writer (4 registers)	777560	3FF70
Console Switch Register	777570	3FF78
11/45 SSR0	777572	3FF7A
11/45 SSR1	777574	3FF7C
11/45 SSR2	777576	3FF7E
User Segmentation Registers		
I Space Descriptor (0-7)	777600	3FF80
D Space Descriptor (0-7)	777620	3FF90

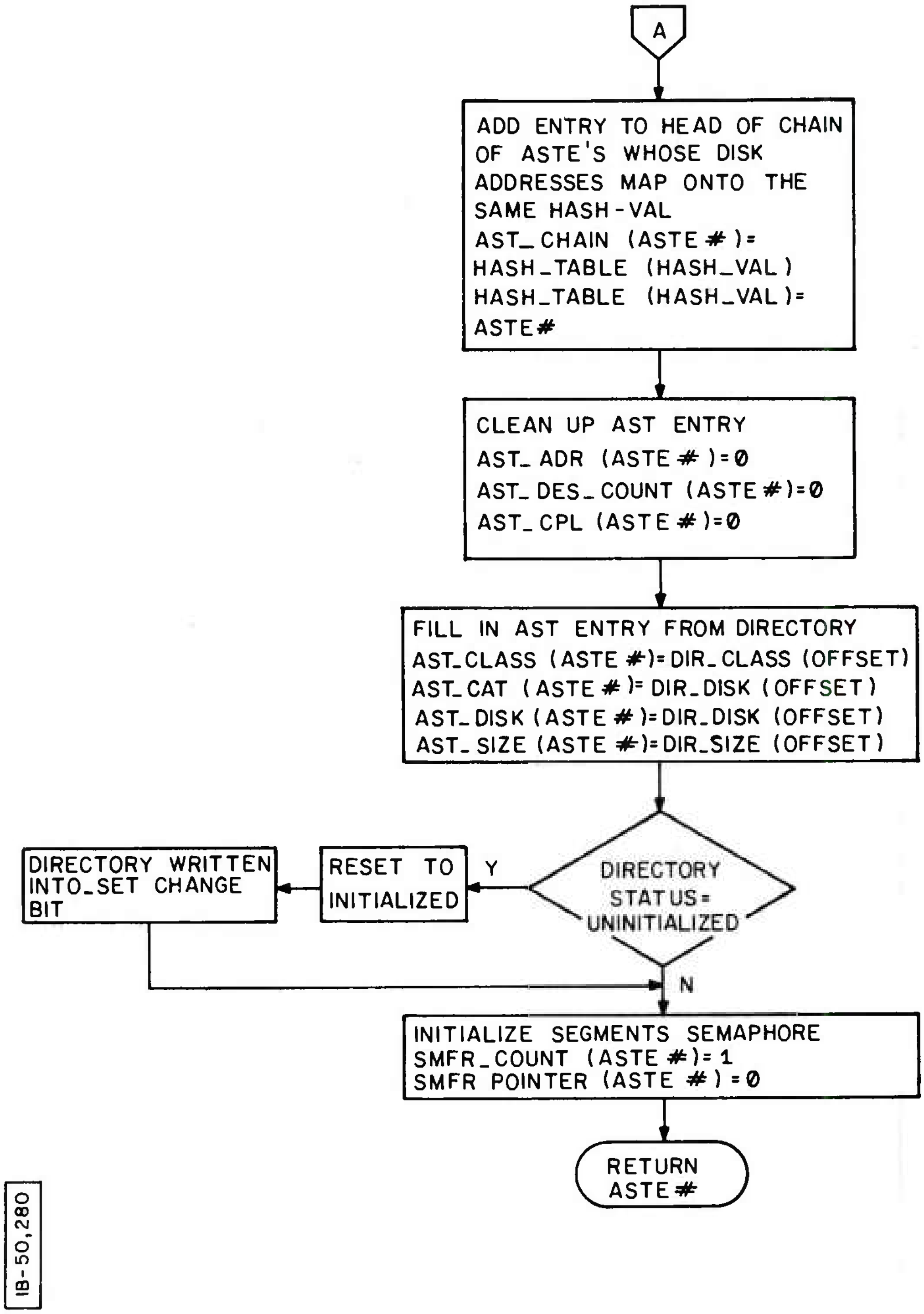
Definition	Octal	Hex
I Space Address (0-7)	777640	3FFA0
D Space Address (0-7)	777660	3FFB0
11/45 PIRQ Register	777772	3FFFA
11/45 Stack Limiter Register	777774	3FFFC
CPU Status (PSW)	777776	3FFFE

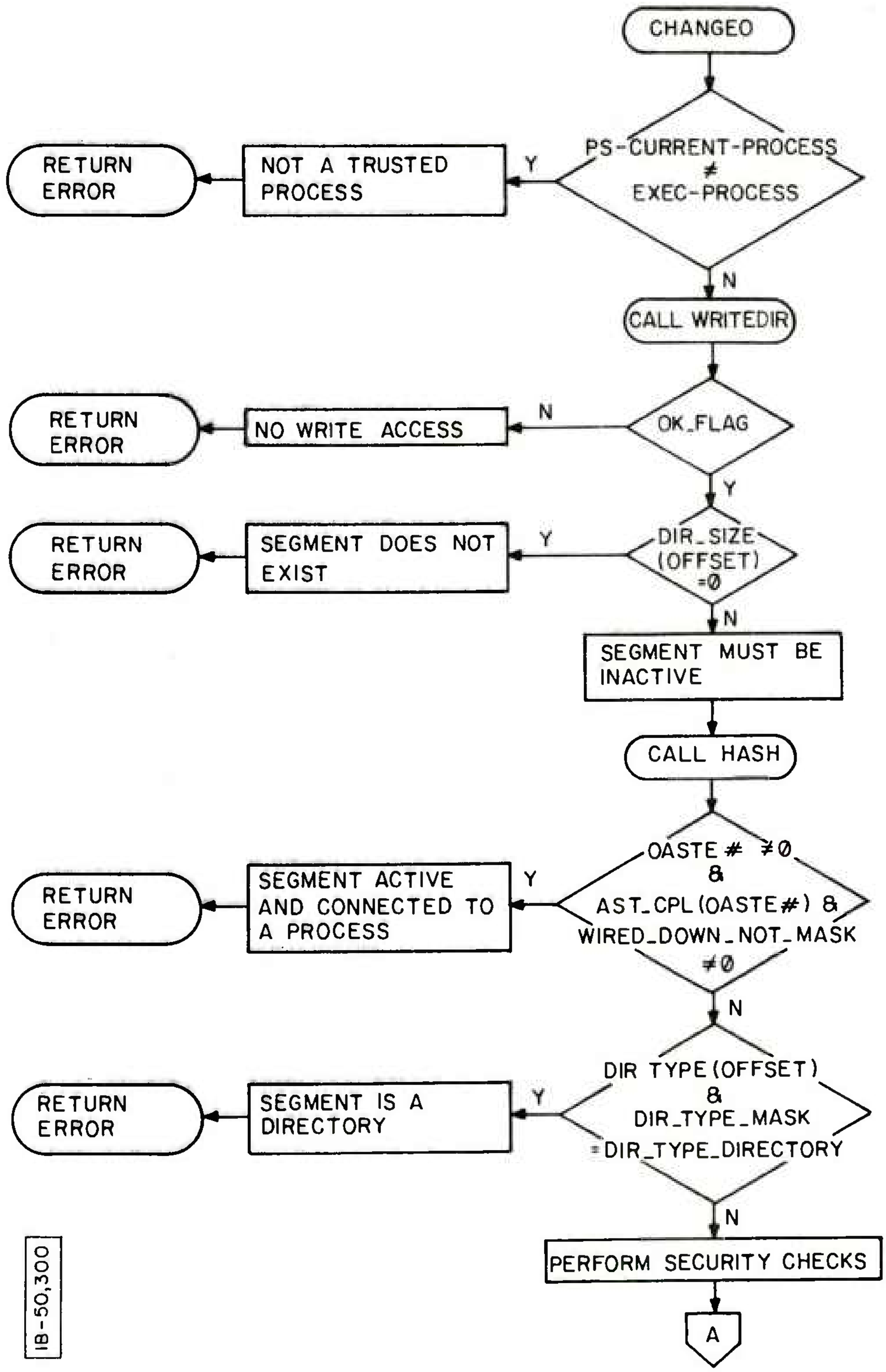
ADDENDA

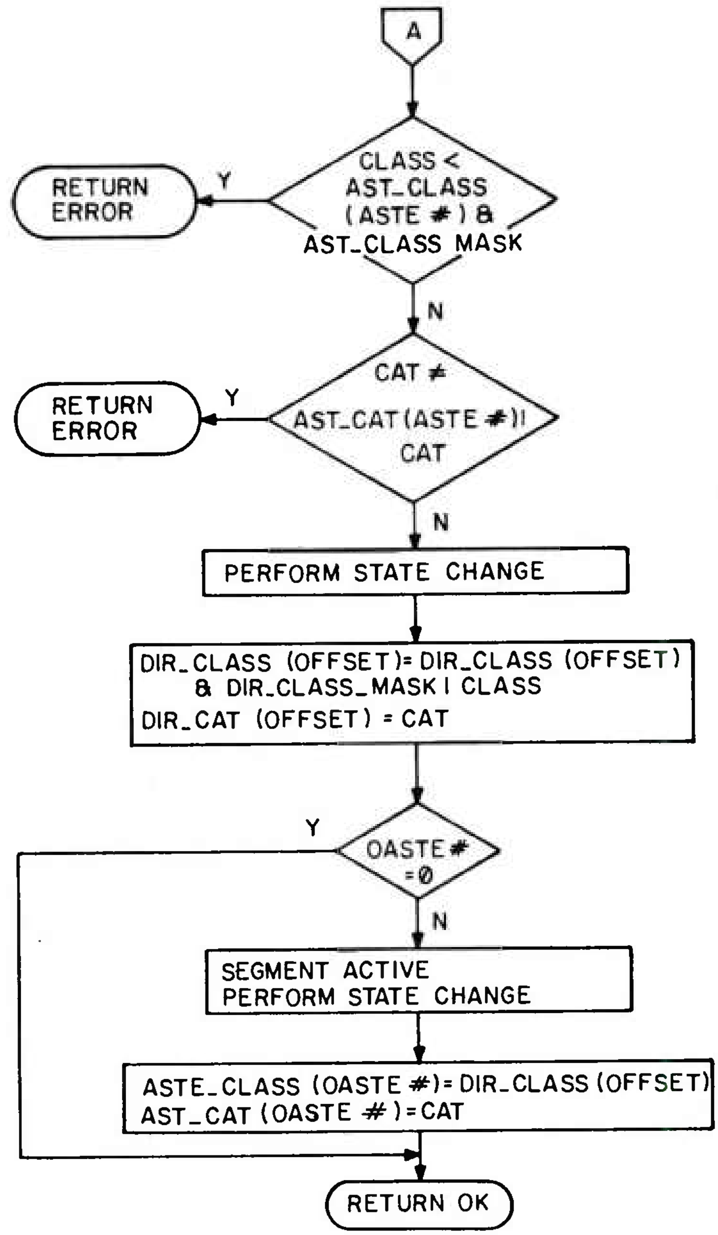
FLOW CHARTS


ACT
AST CHAIN(Ø) =Ø
N
A FREE ASTE # EXISTS
Y
NO ASTE # ON FREE CHAIN: LOOK ON AGE CHAIN FOR "OLDEST" ASTE #
I = Ø
NEXT = AST_AGE_CHAIN (I)
AST_AGE CHAIN (NEXT) =Ø
N
I = NEXT
Y
CALL DEACT
AST ENTRY IS DEACTIVATED: A FREE AST ENTRY EXISTS
ASTE # = AST_CHAIN (Ø)
REMOVE FROM FREE CHAIN
AST_CHAIN (Ø) = AST_CHAIN (ASTE #)
PUT ON AGE CHAIN
AST_AGE_CHAIN (ASTE#) = AST_AGE_CHAIN (Ø)
AST_AGE_CHAIN (Ø) = ASTE #
CALL PREHASH
HASH VALUE ASSOCIATED WITH SEGMENT'S DISK ADDRESS CALCULATED
A

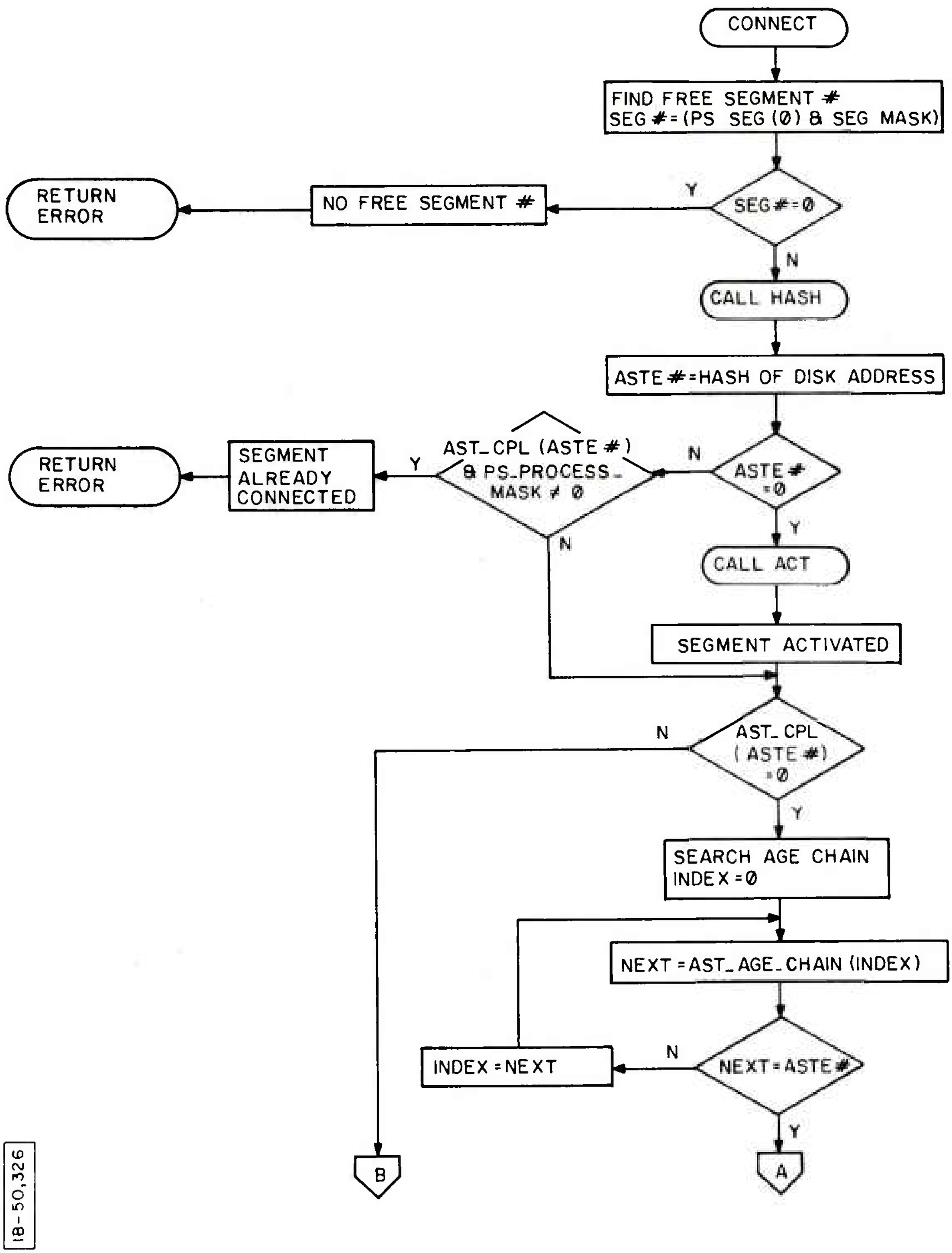
IB-50,279


A
ADD ENTRY TO HEAD OF CHAIN OF ASTE'S WHOSE DISK ADDRESSES MAP ONTO THE SAME HASH-VAL
AST_CHAIN (ASTE#)= HASH_TABLE (HASH_VAL)
HASH_TABLE (HASH_VAL)= ASTE#
CLEAN UP AST ENTRY
AST_ADR (ASTE#)=Ø
AST_DES_COUNT (ASTE#)=Ø
AST_CPL (ASTE#)=Ø
FILL IN AST ENTRY FROM DIRECTORY
AST_CLASS (ASTE#)=DIR_CLASS (OFFSET)
AST_CAT (ASTE#)= DIR_DISK (OFFSET)
AST_DISK (ASTE#)=DIR_DISK (OFFSET)
AST_SIZE (ASTE#)=DIR_SIZE (OFFSET)
DIRECTORY STATUS= UNINITIALIZED
Y
RESET TO INITIALIZED
DIRECTORY WRITTEN INTO_SET CHANGE BIT
N
INITIALIZE SEGMENTS SEMAPHORE
SMFR_COUNT (ASTE#)= 1
SMFR POINTER (ASTE#) = Ø
RETURN ASTE#
1B-50,280

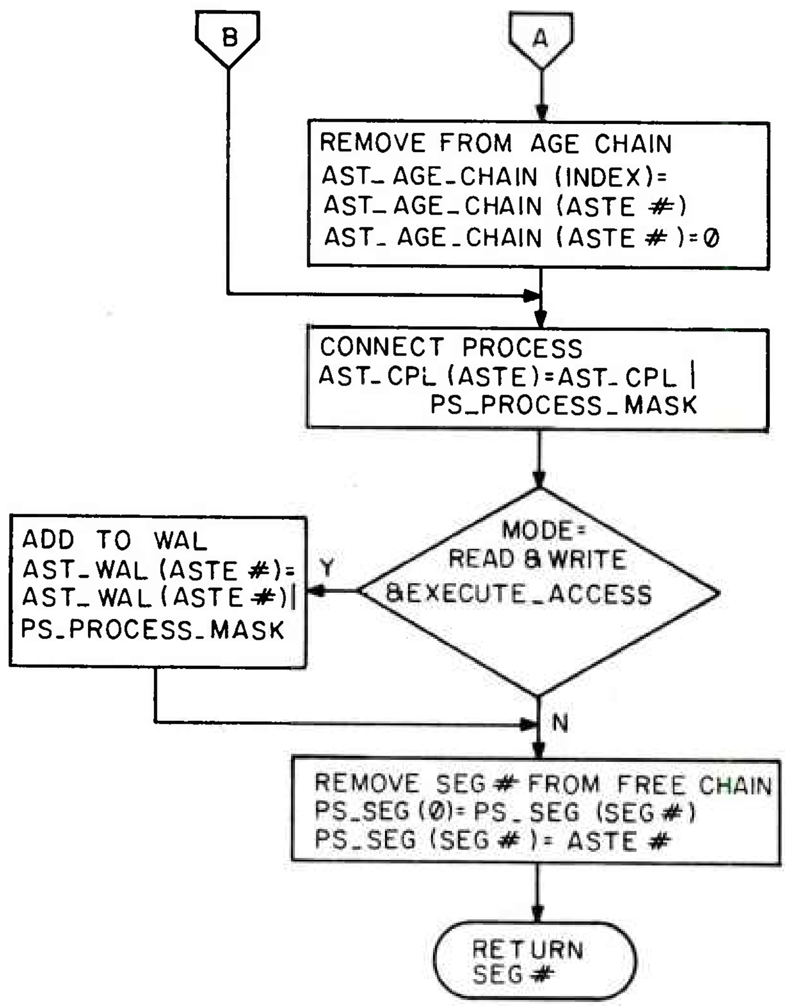

CHANGEO
PS-CURRENT-PROCESS ≠ EXEC-PROCESS
Y
NOT A TRUSTED PROCESS
RETURN ERROR
N
CALL WRITEDIR
OK_FLAG
N
NO WRITE ACCESS
RETURN ERROR
Y
DIR_SIZE (OFFSET) =0
Y
SEGMENT DOES NOT EXIST
RETURN ERROR
N
SEGMENT MUST BE INACTIVE
CALL HASH
OASTE # ≠0 & AST_CPL (OASTE #) & WIRED_DOWN_NOT_MASK ≠0
Y
SEGMENT ACTIVE AND CONNECTED TO A PROCESS
RETURN ERROR
N
DIR TYPE (OFFSET) & DIR_TYPE_MASK =DIR_TYPE_DIRECTORY
Y
SEGMENT IS A DIRECTORY
RETURN ERROR
N
PERFORM SECURITY CHECKS
A
IB-50,300


A
CLASS < AST_CLASS (ASTE #) & AST_CLASS MASK
Y
RETURN ERROR
N
CAT ≠ AST_CAT (ASTE #) I CAT
Y
RETURN ERROR
N
PERFORM STATE CHANGE
DIR_CLASS (OFFSET) = DIR_CLASS (OFFSET) & DIR_CLASS_MASK I CLASS
DIR_CAT (OFFSET) = CAT
OASTE # = Ø
Y
N
SEGMENT ACTIVE
PERFORM STATE CHANGE
ASTE_CLASS (OASTE #) = DIR_CLASS (OFFSET)
AST_CAT (OASTE #) = CAT
RETURN OK

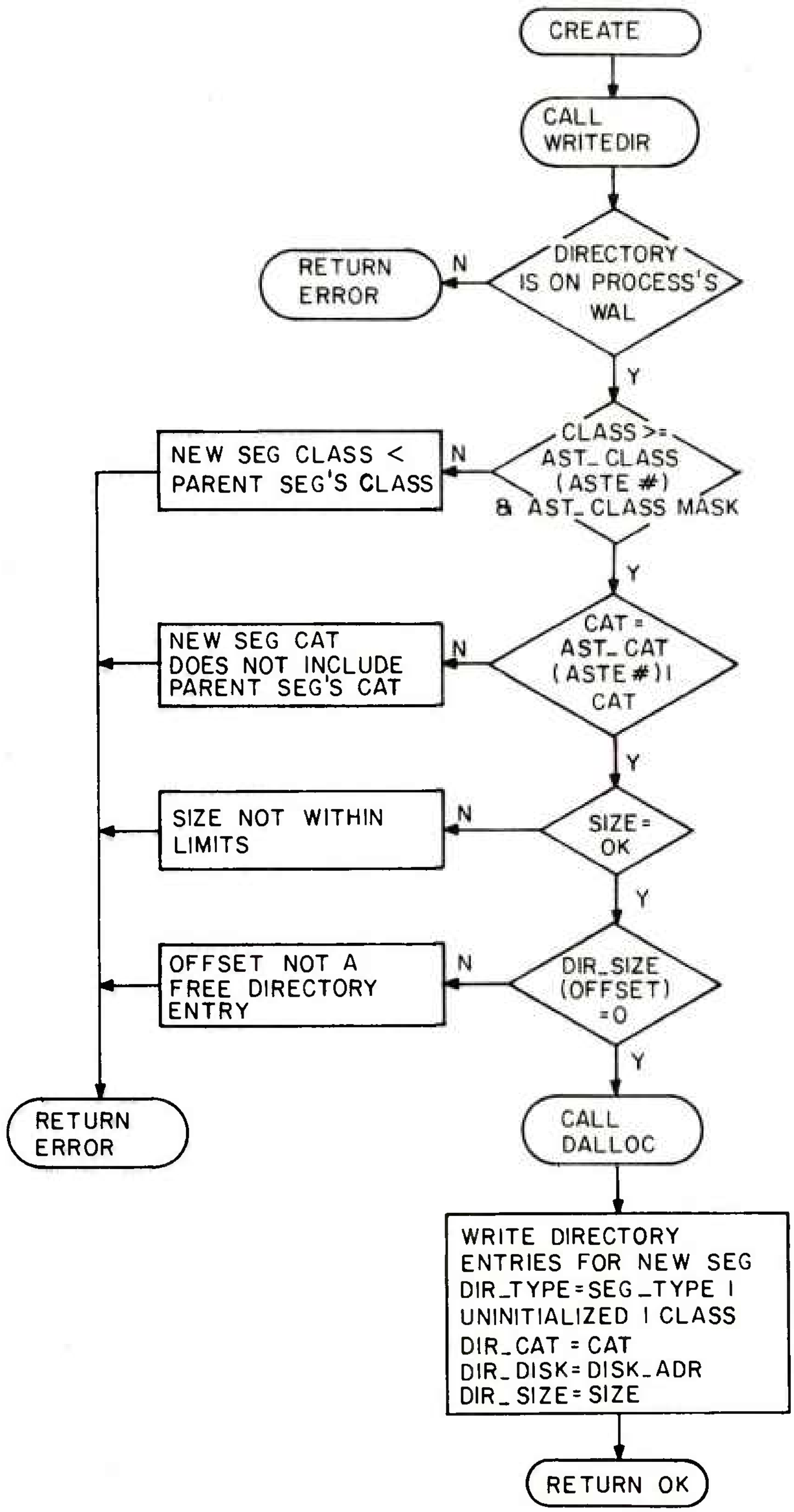
1B-50,301



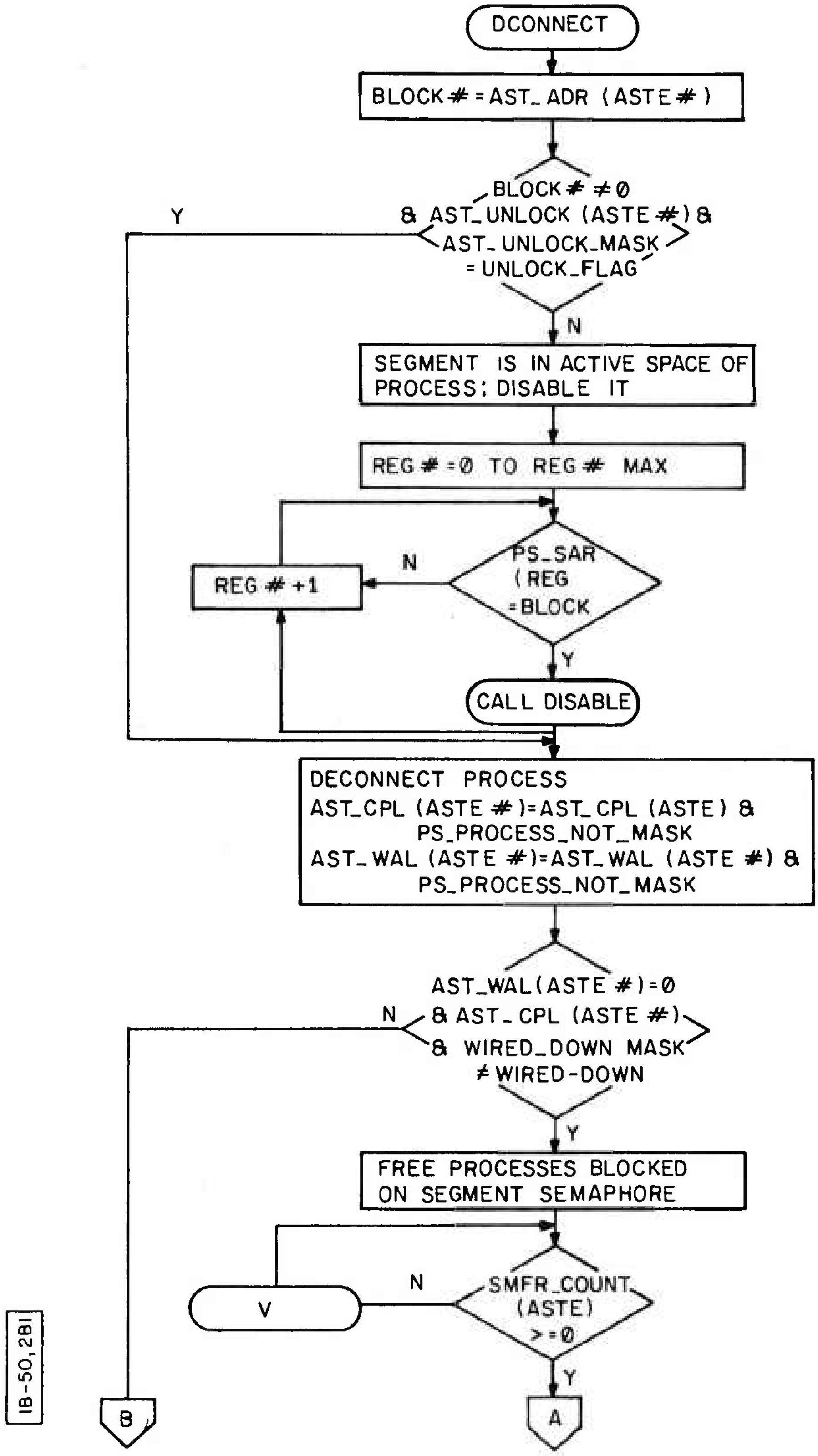
CONNECT
FIND FREE SEGMENT #
SEG # = (PS SEG (Ø) & SEG MASK)
SEG # = Ø
Y
NO FREE SEGMENT #
RETURN ERROR
N
CALL HASH
ASTE # = HASH OF DISK ADDRESS
ASTE # = Ø
N
AST_CPL (ASTE #) & PS_PROCESS_MASK ≠ Ø
Y
SEGMENT ALREADY CONNECTED
RETURN ERROR
N
Y
CALL ACT
SEGMENT ACTIVATED
AST_CPL (ASTE #) = Ø
N
Y
SEARCH AGE CHAIN
INDEX = Ø
NEXT = AST_AGE_CHAIN (INDEX)
NEXT = ASTE #
N
INDEX = NEXT
Y
B
A
18-50,326

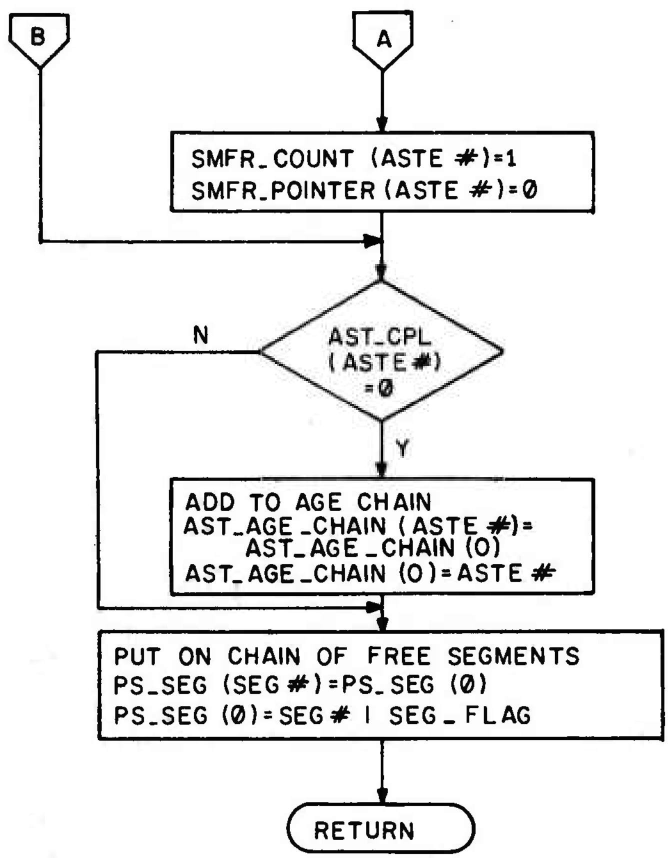

B
A
REMOVE FROM AGE CHAIN
AST_AGE_CHAIN (INDEX)=
AST_AGE_CHAIN (ASTE #)
AST_AGE_CHAIN (ASTE #)=Ø
CONNECT PROCESS
AST_CPL (ASTE)=AST_CPL |
PS_PROCESS_MASK
MODE=
READ & WRITE
&EXECUTE_ACCESS
Y
ADD TO WAL
AST_WAL (ASTE #)=
AST_WAL (ASTE #)|
PS_PROCESS_MASK
N
REMOVE SEG # FROM FREE CHAIN
PS_SEG (Ø)= PS_SEG (SEG #)
PS_SEG (SEG #)= ASTE #
RETURN
SEG #

IB-50,325

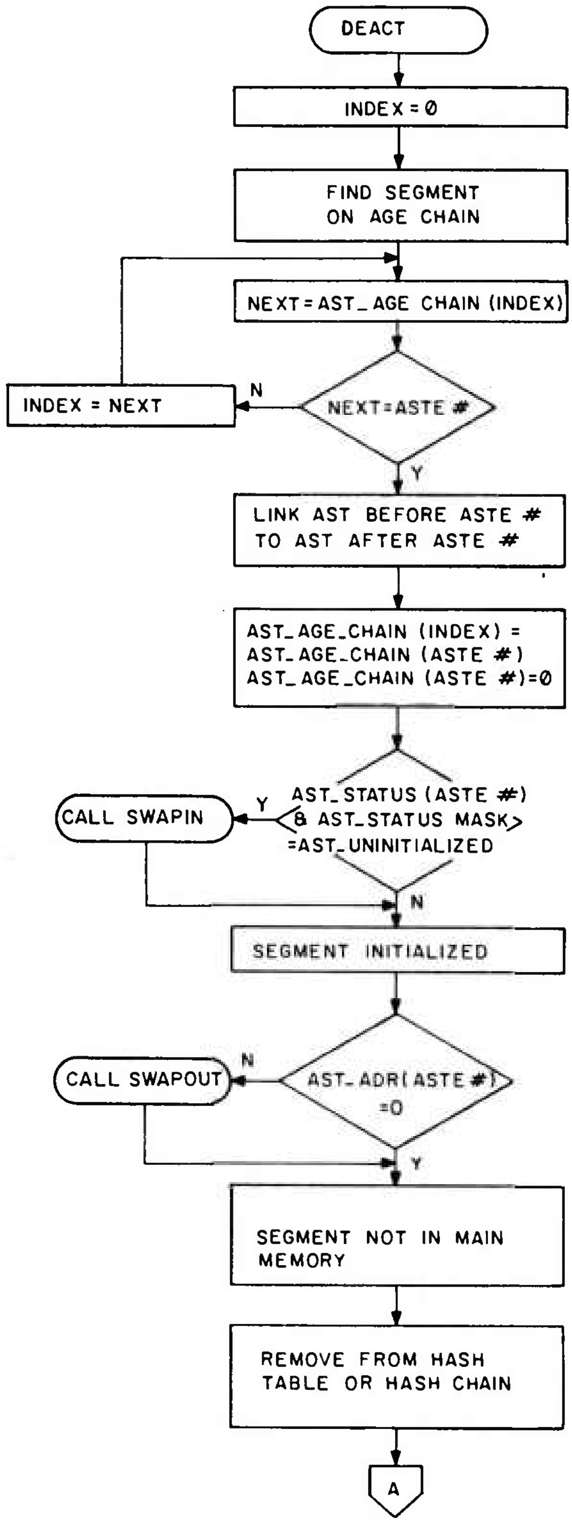

CREATE
CALL WRITEDIR
DIRECTORY IS ON PROCESS'S WAL
N
RETURN ERROR
Y
CLASS >= AST_CLASS (ASTE #) & AST_CLASS MASK
N
NEW SEG CLASS < PARENT SEG'S CLASS
Y
CAT = AST_CAT (ASTE #) I CAT
N
NEW SEG CAT DOES NOT INCLUDE PARENT SEG'S CAT
Y
SIZE = OK
N
SIZE NOT WITHIN LIMITS
Y
DIR_SIZE (OFFSET) = 0
N
OFFSET NOT A FREE DIRECTORY ENTRY
Y
RETURN ERROR
CALL DALLOC
WRITE DIRECTORY ENTRIES FOR NEW SEG
DIR_TYPE=SEG_TYPE I UNINITIALIZED I CLASS
DIR_CAT = CAT
DIR_DISK= DISK_ADR
DIR_SIZE= SIZE
RETURN OK

IB-50,275

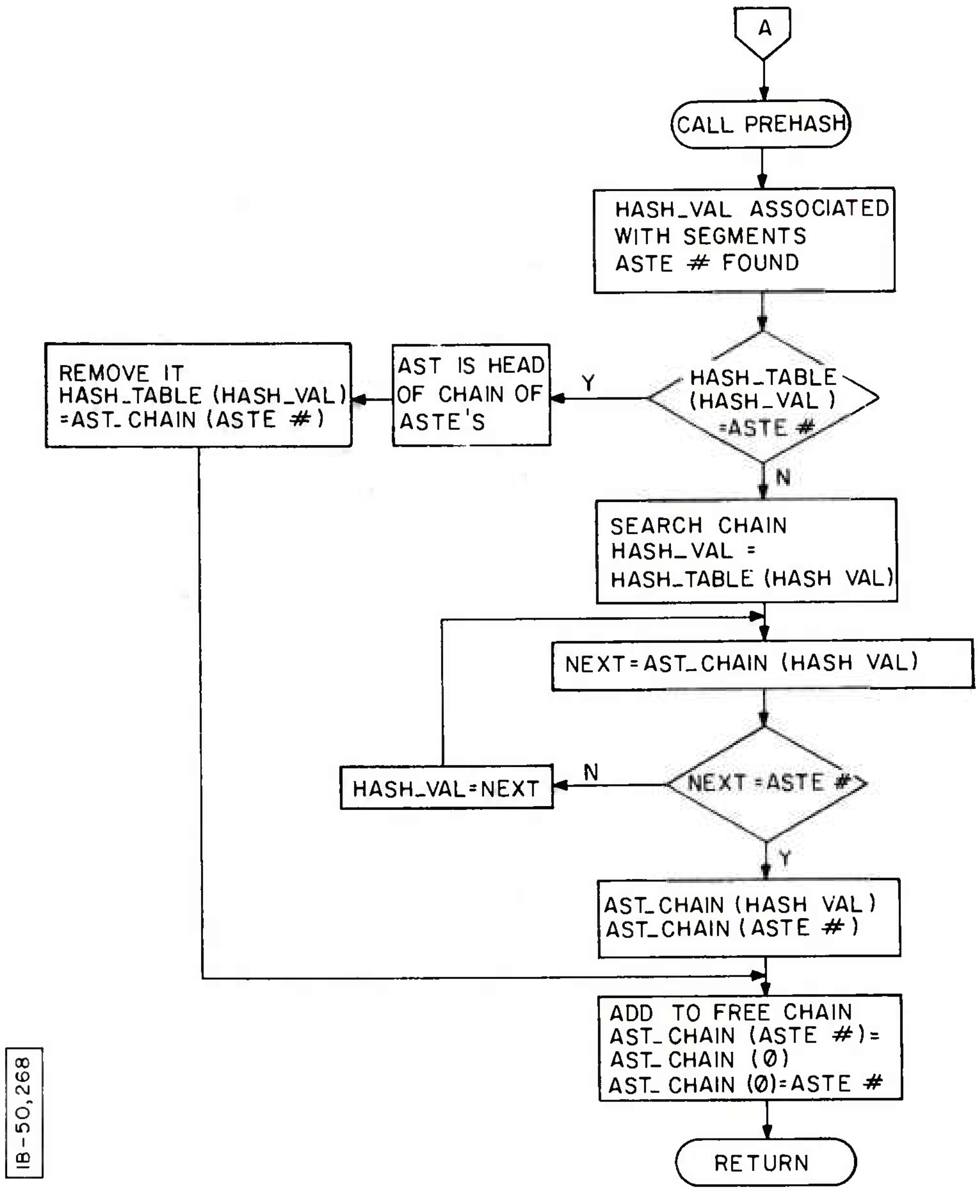

DCONNECT
BLOCK # = AST_ADR (ASTE #)
BLOCK # ≠ Ø & AST_UNLOCK (ASTE #) & AST_UNLOCK_MASK = UNLOCK_FLAG
Y
N
SEGMENT IS IN ACTIVE SPACE OF PROCESS; DISABLE IT
REG # = Ø TO REG # MAX
PS_SAR (REG =BLOCK
N
REG # +1
Y
CALL DISABLE
DECONNECT PROCESS
AST_CPL (ASTE #)=AST_CPL (ASTE) & PS_PROCESS_NOT_MASK
AST_WAL (ASTE #)=AST_WAL (ASTE #) & PS_PROCESS_NOT_MASK
AST_WAL(ASTE #) = Ø & AST_CPL (ASTE #) & WIRED_DOWN MASK ≠ WIRED-DOWN
N
Y
FREE PROCESSES BLOCKED ON SEGMENT SEMAPHORE
SMFR_COUNT (ASTE) >= Ø
N
V
Y
B
A
IB-50,281


B
A
SMFR_COUNT (ASTE #)=1
SMFR_POINTER (ASTE #)=Ø
N
AST_CPL (ASTE #) =Ø
Y
ADD TO AGE CHAIN
AST_AGE_CHAIN (ASTE #)= AST_AGE_CHAIN (O)
AST_AGE_CHAIN (O)=ASTE #
PUT ON CHAIN OF FREE SEGMENTS
PS_SEG (SEG #)=PS_SEG (Ø)
PS_SEG (Ø)=SEG # I SEG_FLAG
RETURN

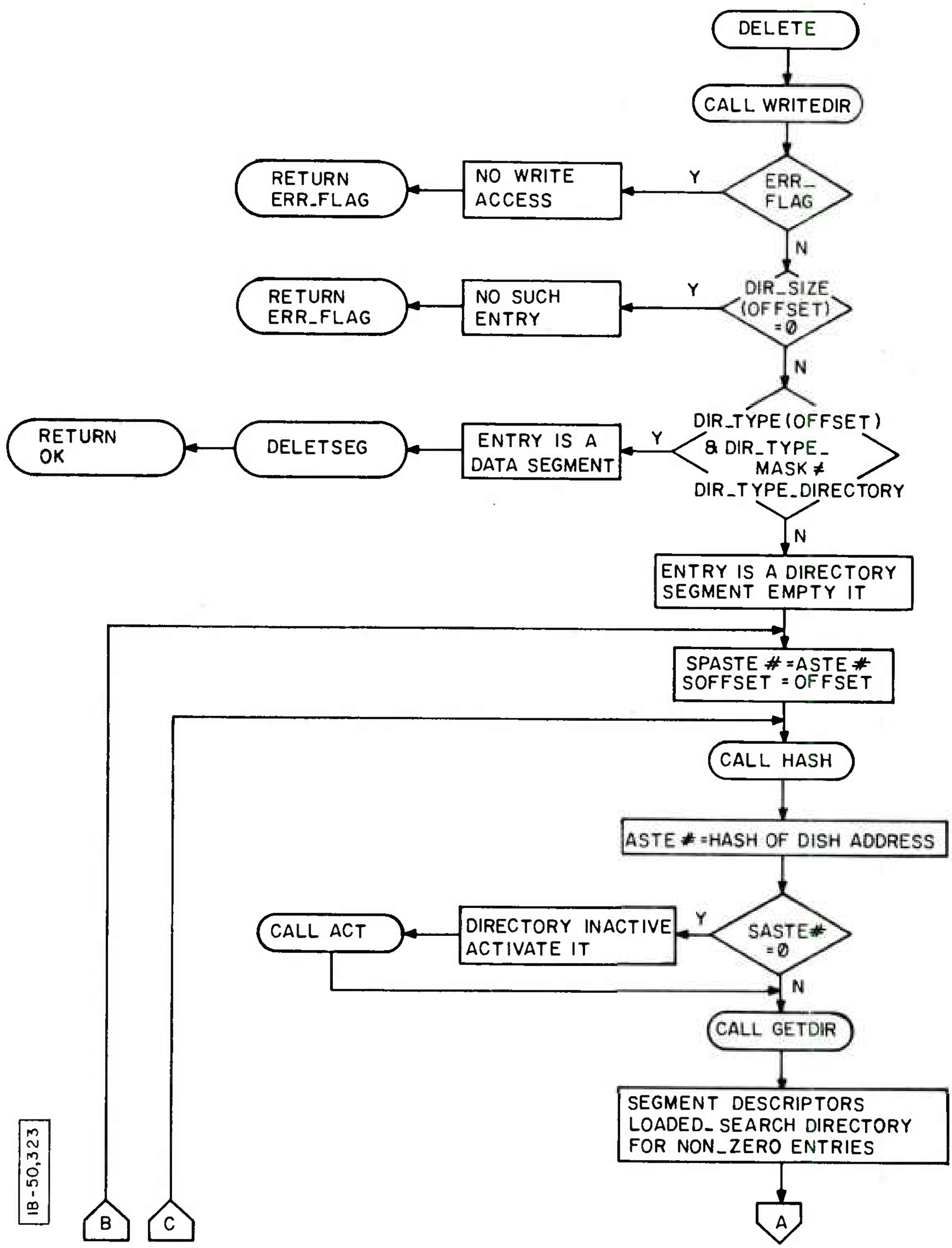
IB-50,282


DEACT
INDEX = Ø
FIND SEGMENT ON AGE CHAIN
NEXT = AST_AGE CHAIN (INDEX)
INDEX = NEXT
N
NEXT = ASTE #
Y
LINK AST BEFORE ASTE # TO AST AFTER ASTE #
AST_AGE_CHAIN (INDEX) = AST_AGE_CHAIN (ASTE #) AST_AGE_CHAIN (ASTE #)=Ø
CALL SWAPIN
Y
AST_STATUS (ASTE #) & AST_STATUS MASK =AST_UNINITIALIZED
N
SEGMENT INITIALIZED
CALL SWAPOUT
N
AST_ADR(ASTE #) =0
Y
SEGMENT NOT IN MAIN MEMORY
REMOVE FROM HASH TABLE OR HASH CHAIN
A

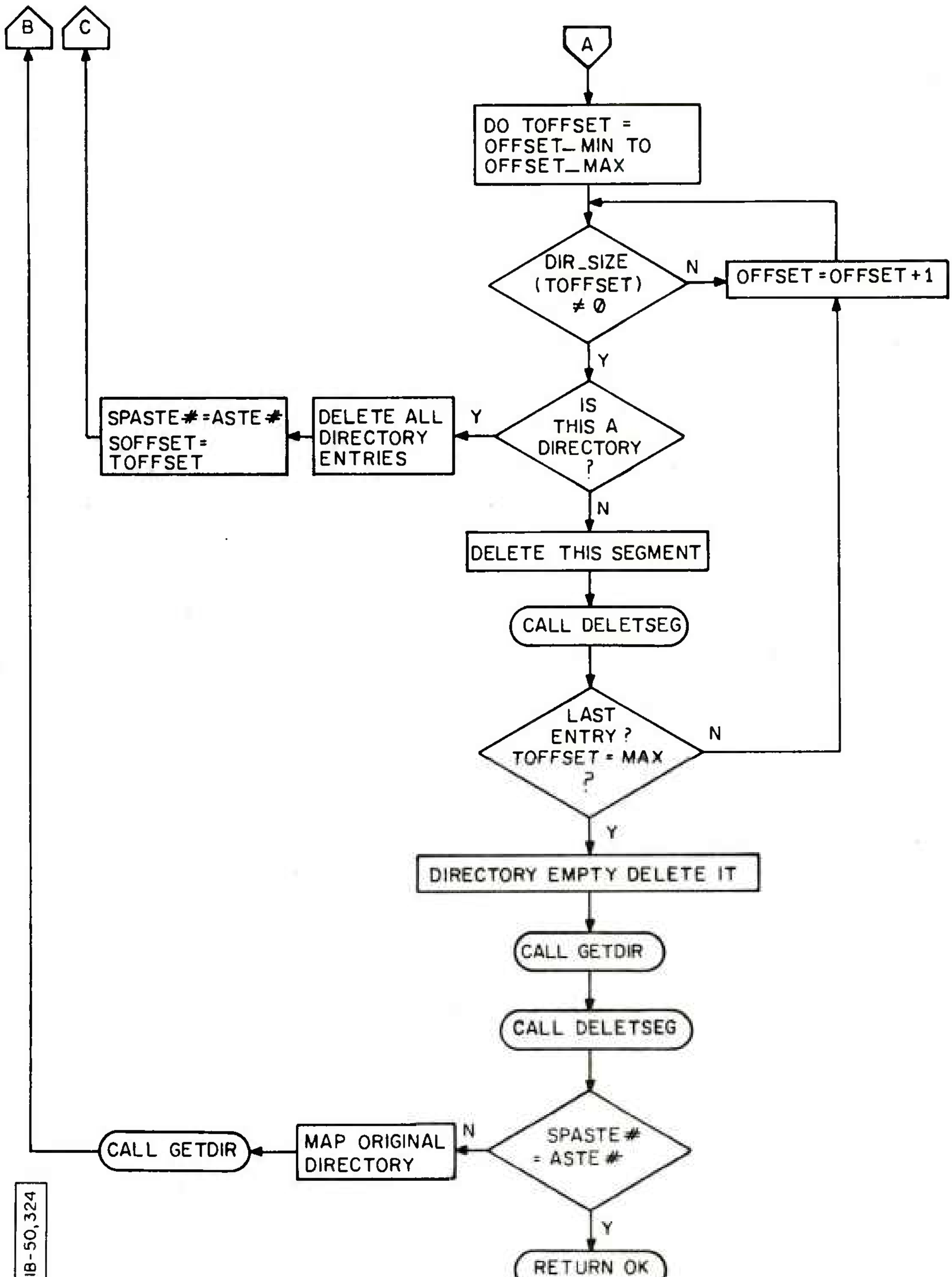
IB-50,267

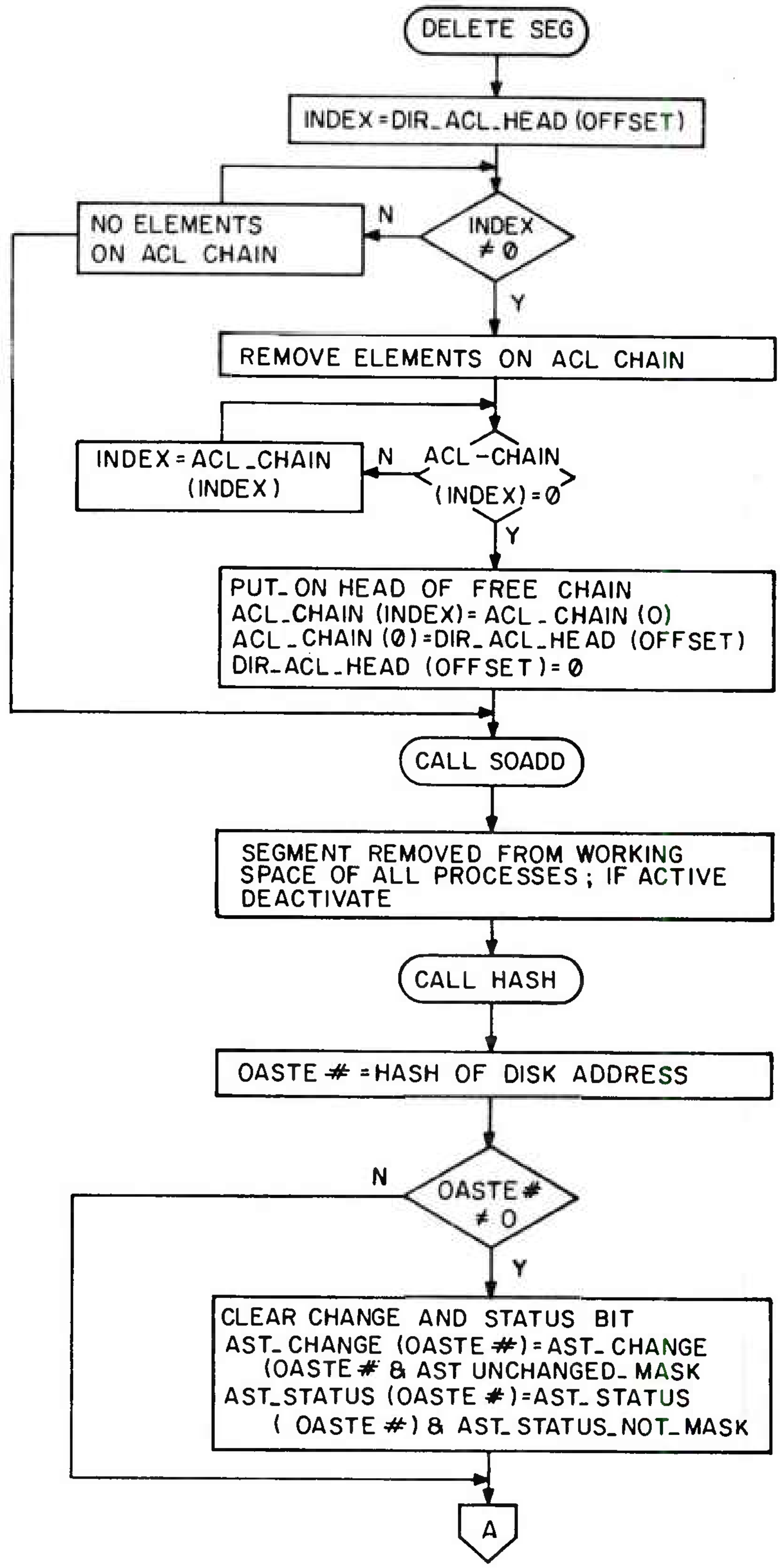


A
CALL PREHASH
HASH_VAL ASSOCIATED WITH SEGMENTS ASTE # FOUND
HASH_TABLE (HASH_VAL) =ASTE #
Y
N
AST IS HEAD OF CHAIN OF ASTE'S
REMOVE IT HASH_TABLE (HASH_VAL) =AST_CHAIN (ASTE #)
SEARCH CHAIN HASH_VAL = HASH_TABLE (HASH VAL)
NEXT=AST_CHAIN (HASH VAL)
NEXT=ASTE #
N
Y
HASH_VAL=NEXT
AST_CHAIN (HASH VAL) AST_CHAIN (ASTE #)
ADD TO FREE CHAIN AST_CHAIN (ASTE #)= AST_CHAIN (0) AST_CHAIN (0)=ASTE #
RETURN
IB-50,268

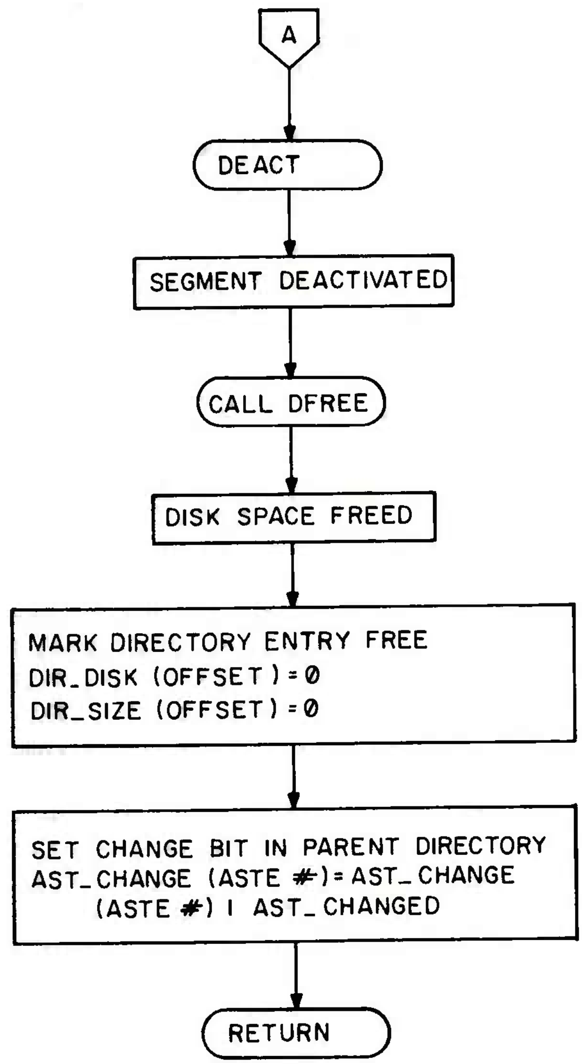


DELETE
CALL WRITEDIR
ERR_FLAG
Y
N
NO WRITE ACCESS
RETURN ERR_FLAG
DIR_SIZE (OFFSET) =Ø
NO SUCH ENTRY
RETURN ERR_FLAG
DIR_TYPE (OFFSET) & DIR_TYPE_MASK ≠ DIR_TYPE_DIRECTORY
ENTRY IS A DATA SEGMENT
DELETSEG
RETURN OK
ENTRY IS A DIRECTORY SEGMENT EMPTY IT
SPASTE # =ASTE #
SOFFSET = OFFSET
CALL HASH
ASTE # =HASH OF DISH ADDRESS
SASTE# =Ø
DIRECTORY INACTIVE ACTIVATE IT
CALL ACT
CALL GETDIR
SEGMENT DESCRIPTORS LOADED_ SEARCH DIRECTORY FOR NON_ZERO ENTRIES
A
B
C
IB-50,323

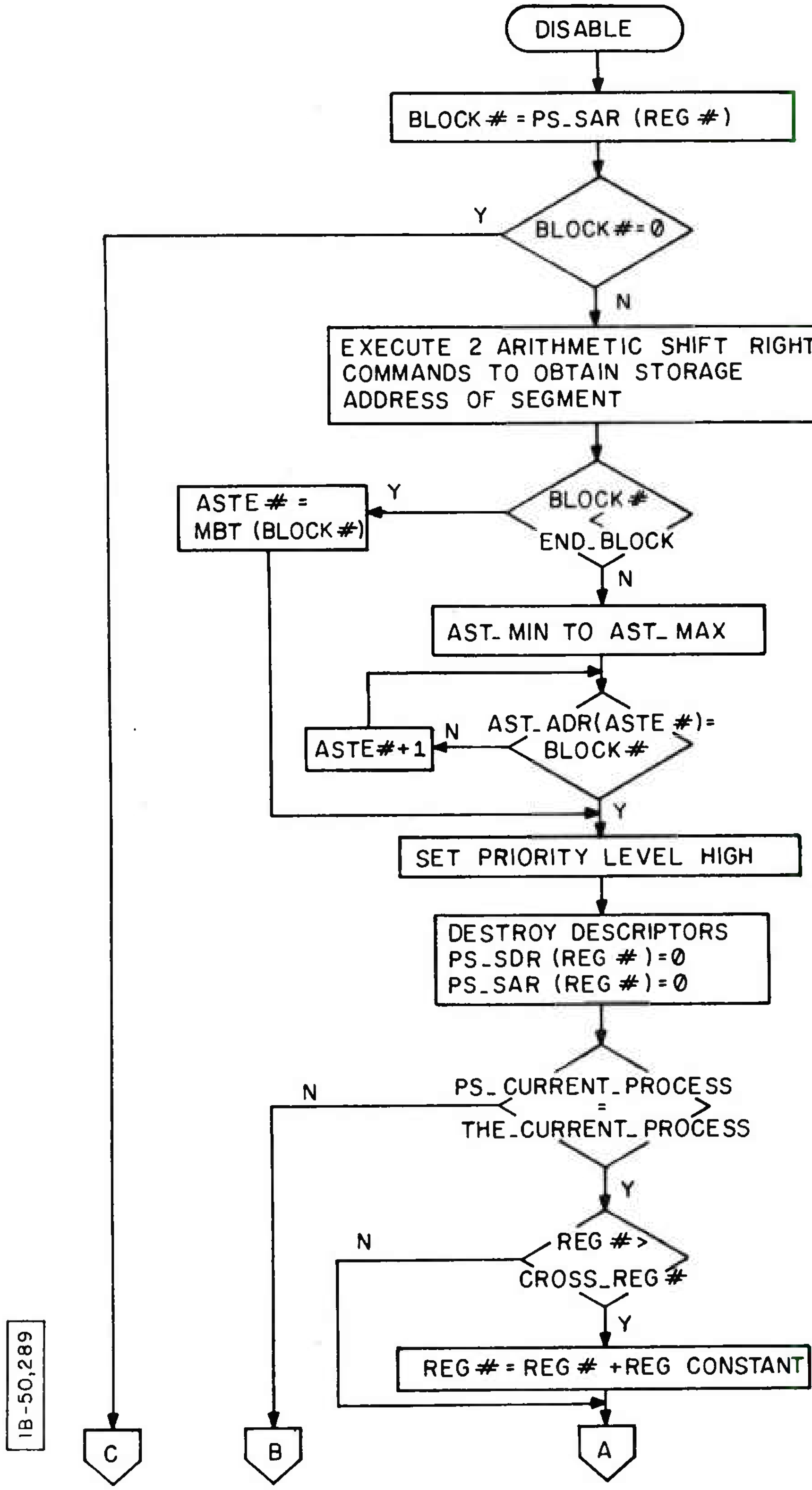

B
C
A
DO TOFFSET =
OFFSET_MIN TO
OFFSET_MAX
DIR_SIZE
(TOFFSET)
≠ Ø
N
OFFSET = OFFSET + 1
Y
IS
THIS A
DIRECTORY
?
Y
DELETE ALL
DIRECTORY
ENTRIES
SPASTE # = ASTE #
SOFFSET =
TOFFSET
N
DELETE THIS SEGMENT
CALL DELETSEG
LAST
ENTRY ?
TOFFSET = MAX
?
N
Y
DIRECTORY EMPTY DELETE IT
CALL GETDIR
CALL DELETSEG
SPASTE #
= ASTE #
N
MAP ORIGINAL
DIRECTORY
CALL GETDIR
Y
RETURN OK
IB-50,324

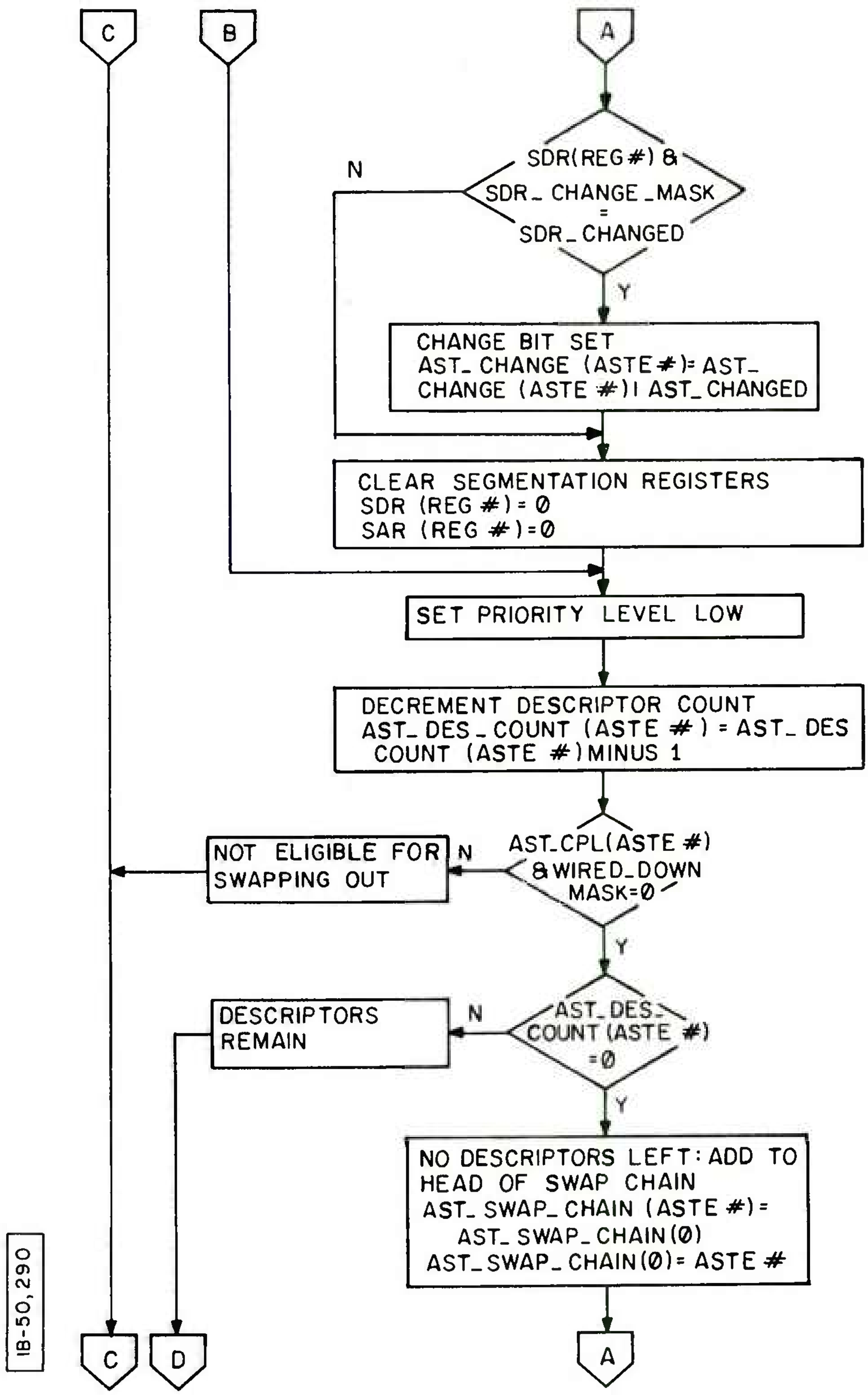

DELETE SEG
INDEX = DIR_ACL_HEAD (OFFSET)
INDEX ≠ 0
N
Y
NO ELEMENTS ON ACL CHAIN
REMOVE ELEMENTS ON ACL CHAIN
ACL-CHAIN (INDEX) = 0
N
Y
INDEX = ACL_CHAIN (INDEX)
PUT_ON HEAD OF FREE CHAIN
ACL_CHAIN (INDEX) = ACL_CHAIN (0)
ACL_CHAIN (0) = DIR_ACL_HEAD (OFFSET)
DIR_ACL_HEAD (OFFSET) = 0
CALL SOADD
SEGMENT REMOVED FROM WORKING SPACE OF ALL PROCESSES; IF ACTIVE DEACTIVATE
CALL HASH
OASTE # = HASH OF DISK ADDRESS
OASTE # ≠ 0
N
Y
CLEAR CHANGE AND STATUS BIT
AST_CHANGE (OASTE #) = AST_CHANGE (OASTE # & AST UNCHANGED_MASK
AST_STATUS (OASTE #) = AST_STATUS (OASTE #) & AST_STATUS_NOT_MASK
A

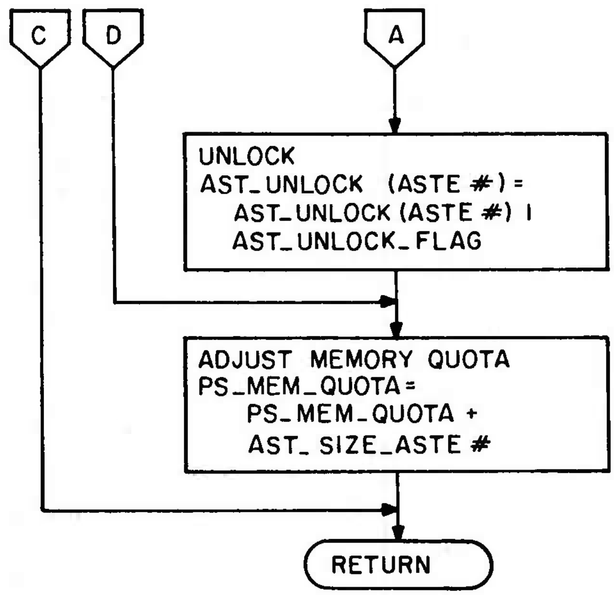
IB-50,292


A
DEACT
SEGMENT DEACTIVATED
CALL DFREE
DISK SPACE FREED
MARK DIRECTORY ENTRY FREE
DIR_DISK (OFFSET) = 0
DIR_SIZE (OFFSET) = 0
SET CHANGE BIT IN PARENT DIRECTORY
AST_CHANGE (ASTE #) = AST_CHANGE (ASTE #) | AST_CHANGED
RETURN

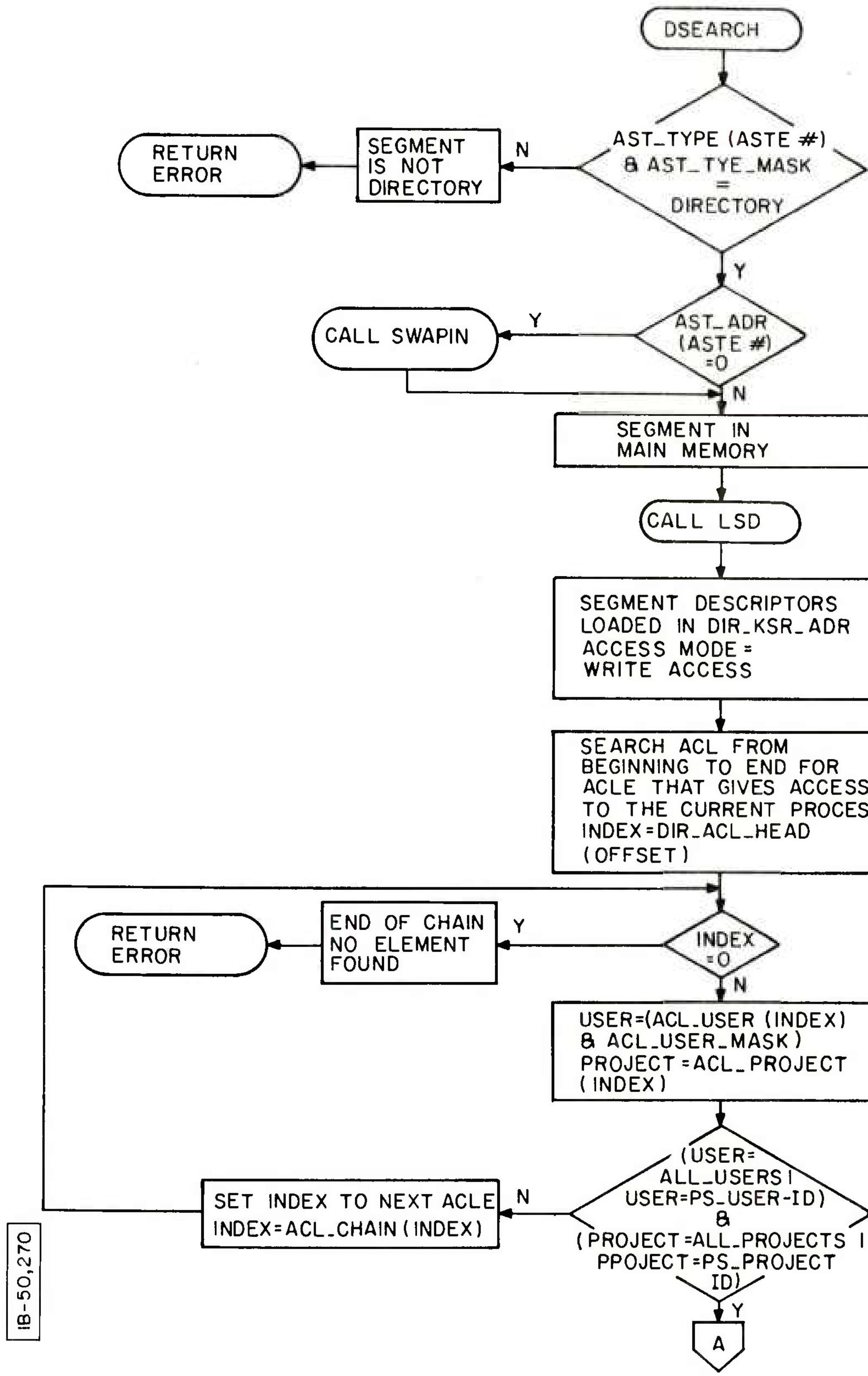
IB-50,293

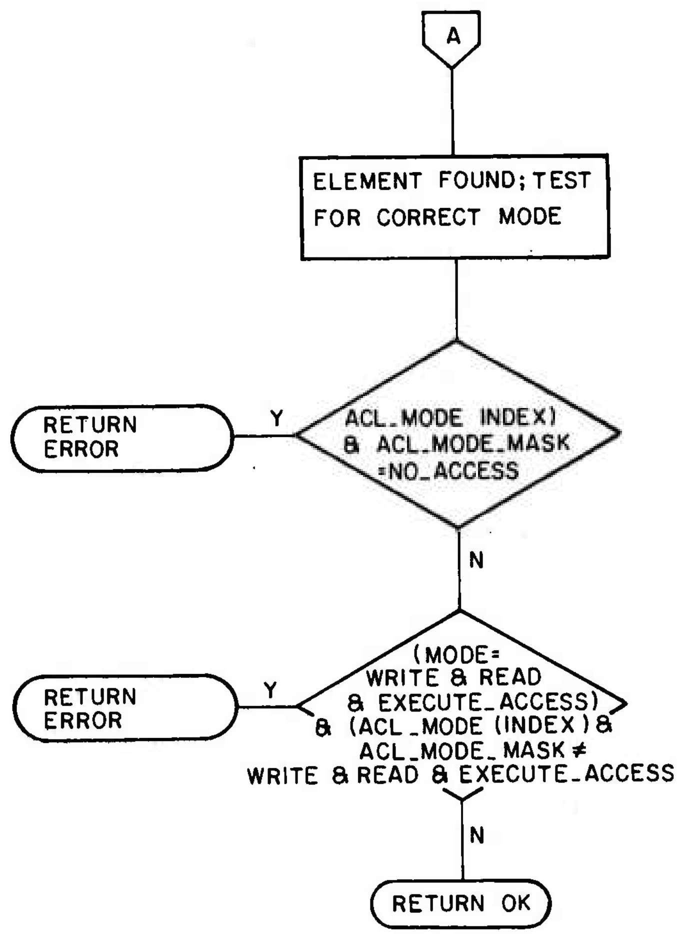

DISABLE
BLOCK # = PS_SAR (REG #)
BLOCK # = Ø
Y
N
EXECUTE 2 ARITHMETIC SHIFT RIGHT COMMANDS TO OBTAIN STORAGE ADDRESS OF SEGMENT
BLOCK # < END_BLOCK
Y
ASTE # = MBT (BLOCK #)
N
AST_MIN TO AST_MAX
AST_ADR(ASTE #) = BLOCK #
N
ASTE # + 1
Y
SET PRIORITY LEVEL HIGH
DESTROY DESCRIPTORS
PS_SDR (REG #) = Ø
PS_SAR (REG #) = Ø
PS_CURRENT_PROCESS = THE_CURRENT_PROCESS
N
Y
REG # > CROSS_REG #
N
Y
REG # = REG # + REG CONSTANT
C
B
A
IB-50,289


A
B
C
SDR(REG #) &
SDR_ CHANGE_MASK
=
SDR_CHANGED
N
Y
CHANGE BIT SET
AST_ CHANGE (ASTE #)= AST_
CHANGE (ASTE #)| AST_CHANGED
CLEAR SEGMENTATION REGISTERS
SDR (REG #) = Ø
SAR (REG #) = Ø
SET PRIORITY LEVEL LOW
DECREMENT DESCRIPTOR COUNT
AST_ DES_ COUNT (ASTE #) = AST_ DES
COUNT (ASTE #) MINUS 1
AST_CPL(ASTE #)
& WIRED_DOWN
MASK=Ø
N
NOT ELIGIBLE FOR
SWAPPING OUT
Y
AST_DES_
COUNT (ASTE #)
=Ø
N
DESCRIPTORS
REMAIN
Y
NO DESCRIPTORS LEFT: ADD TO
HEAD OF SWAP CHAIN
AST_ SWAP_ CHAIN (ASTE #) =
AST_ SWAP_ CHAIN (Ø)
AST_SWAP_ CHAIN (Ø)= ASTE #
C
D
A
IB-50,290

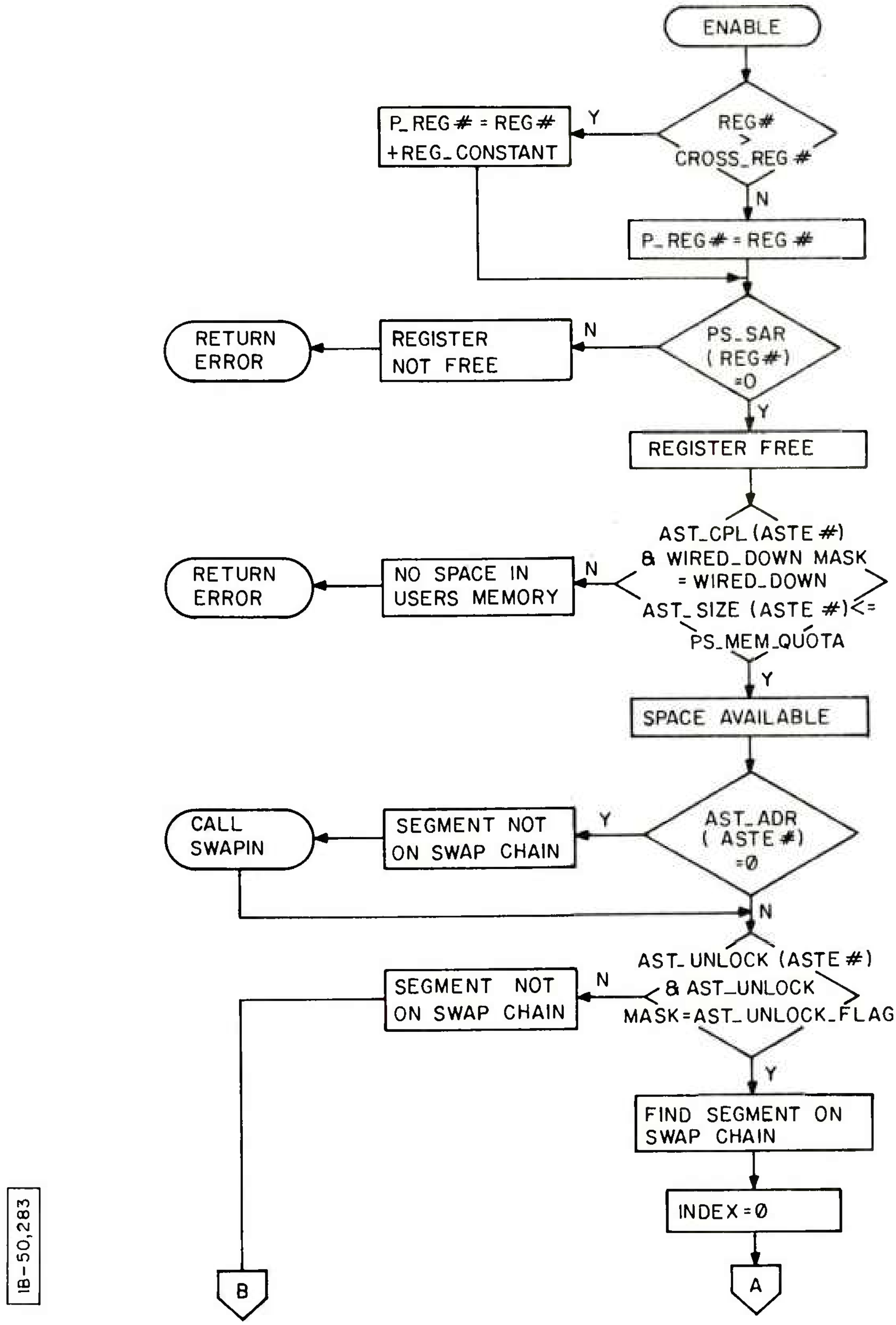

C
D
A
UNLOCK
AST_UNLOCK (ASTE #) =
AST_UNLOCK (ASTE #) I
AST_UNLOCK_FLAG
ADJUST MEMORY QUOTA
PS_MEM_QUOTA =
PS_MEM_QUOTA +
AST_ SIZE_ASTE #
RETURN

18-50,291

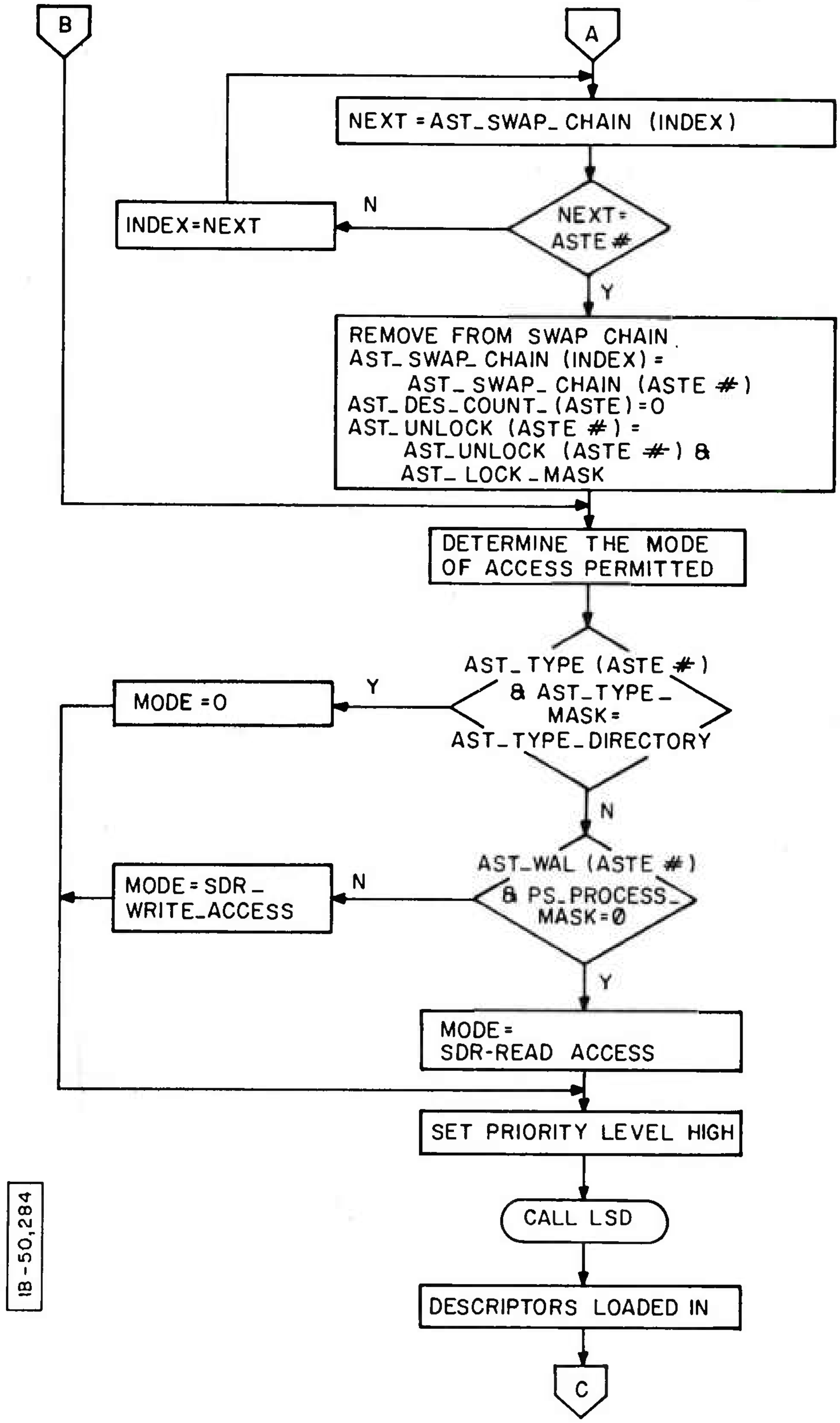

DSEARCH
AST_TYPE (ASTE #) & AST_TYE_MASK = DIRECTORY
N
SEGMENT IS NOT DIRECTORY
RETURN ERROR
Y
AST_ADR (ASTE #) =0
Y
CALL SWAPIN
N
SEGMENT IN MAIN MEMORY
CALL LSD
SEGMENT DESCRIPTORS LOADED IN DIR_KSR_ADR ACCESS MODE = WRITE ACCESS
SEARCH ACL FROM BEGINNING TO END FOR ACLE THAT GIVES ACCESS TO THE CURRENT PROCESS INDEX=DIR_ACL_HEAD (OFFSET)
INDEX =0
Y
END OF CHAIN NO ELEMENT FOUND
RETURN ERROR
N
USER=(ACL_USER (INDEX) & ACL_USER_MASK) PROJECT=ACL_PROJECT (INDEX)
(USER= ALL_USERS \| USER=PS_USER-ID) & (PROJECT=ALL_PROJECTS \| PPOJECT=PS_PROJECT ID)
N
SET INDEX TO NEXT ACLE INDEX=ACL_CHAIN (INDEX)
Y
A
IB-50,270

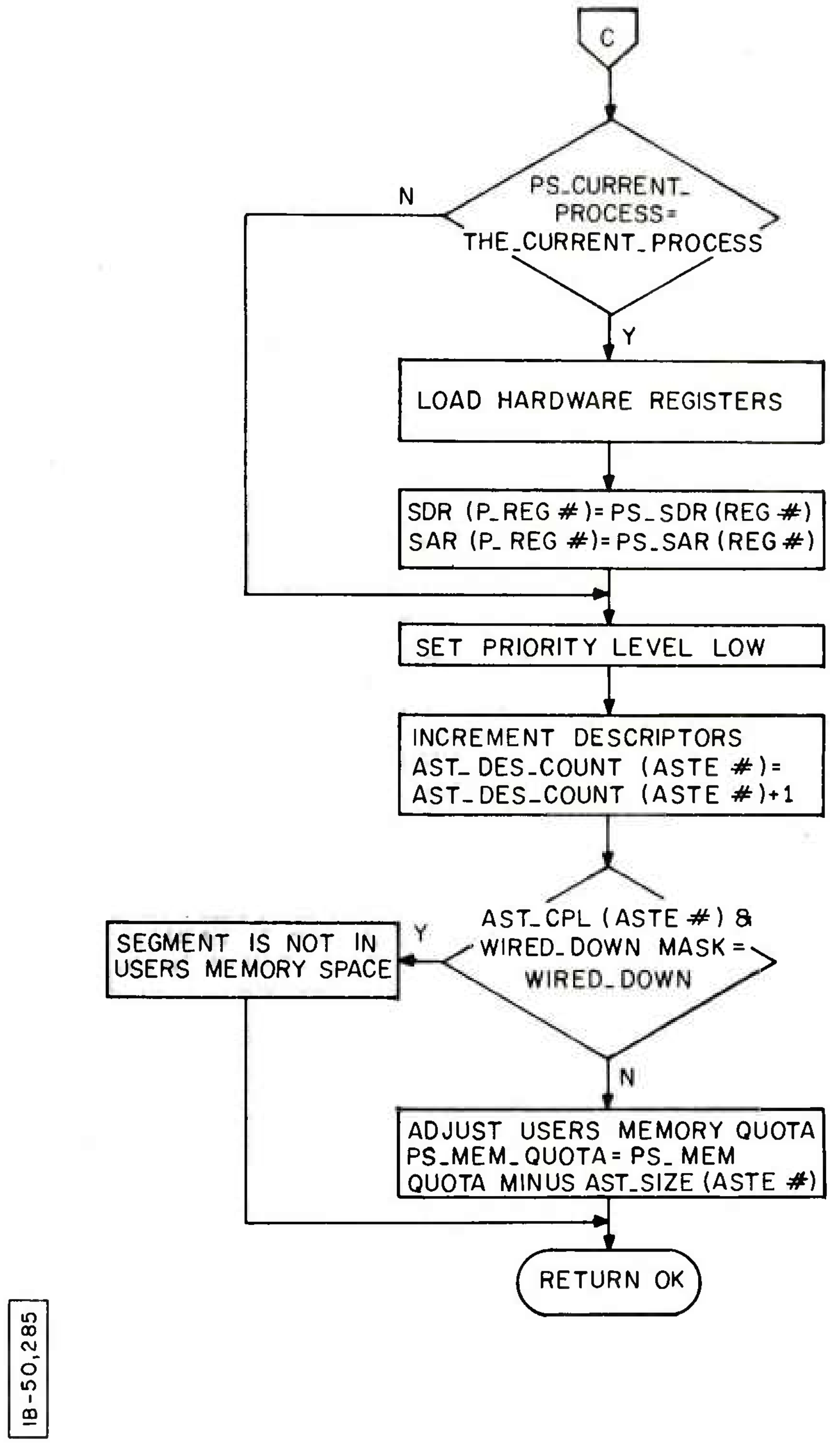

A
ELEMENT FOUND; TEST FOR CORRECT MODE
RETURN ERROR
Y
ACL_MODE INDEX) & ACL_MODE_MASK =NO_ACCESS
N
RETURN ERROR
Y
(MODE= WRITE & READ & EXECUTE_ACCESS) & (ACL_MODE (INDEX) & ACL_MODE_MASK ≠ WRITE & READ & EXECUTE_ACCESS
N
RETURN OK

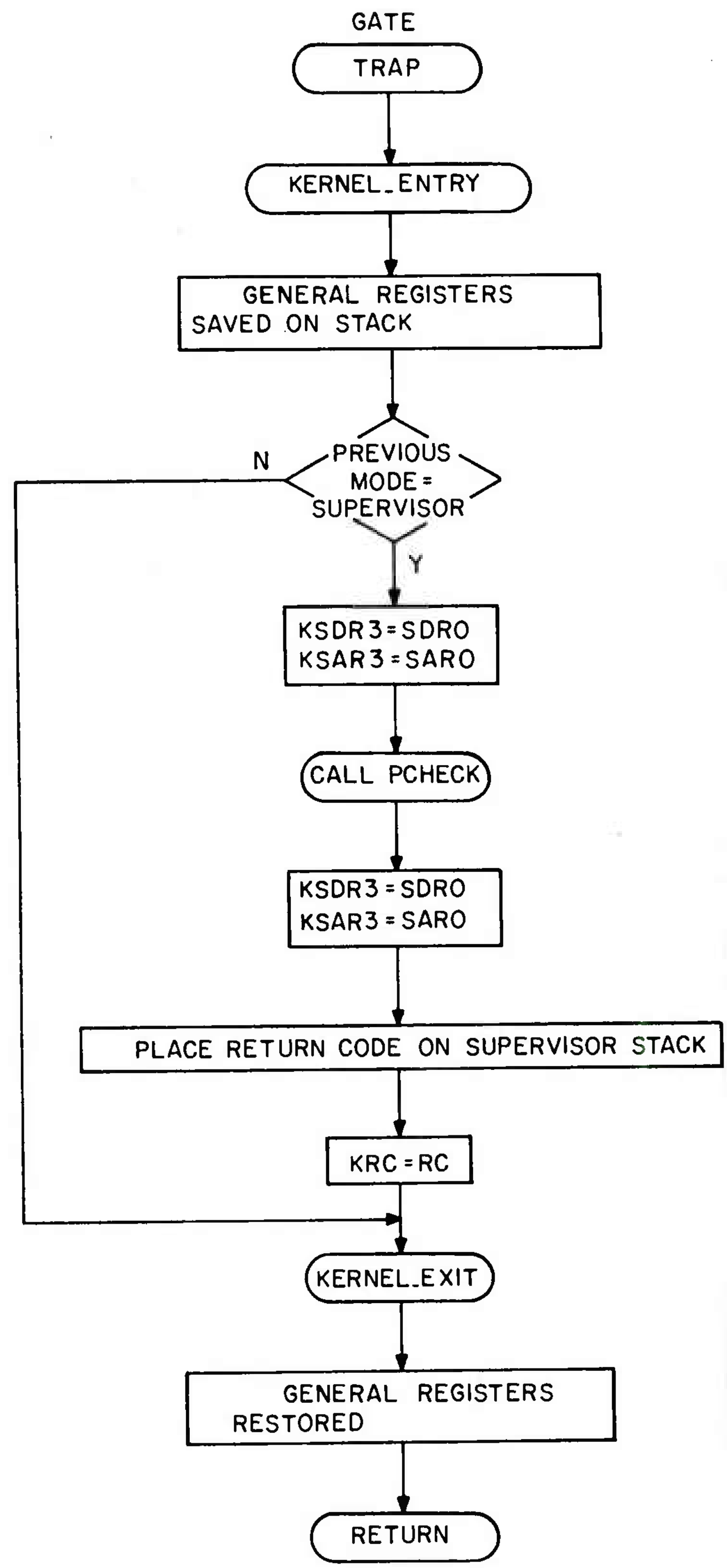
IB-50,269



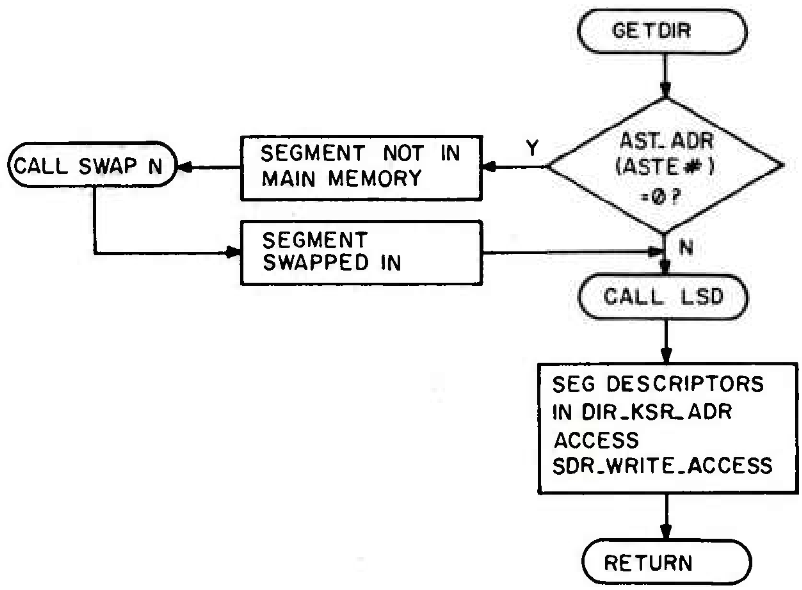
ENABLE
REG# > CROSS_REG #
Y
N
P_REG # = REG # +REG_CONSTANT
P_REG # = REG #
PS_SAR (REG#) =0
N
Y
REGISTER NOT FREE
RETURN ERROR
REGISTER FREE
AST_CPL (ASTE #) & WIRED_DOWN MASK = WIRED_DOWN
AST_SIZE (ASTE #)<= PS_MEM_QUOTA
N
Y
NO SPACE IN USERS MEMORY
RETURN ERROR
SPACE AVAILABLE
AST_ADR (ASTE #) =Ø
Y
N
SEGMENT NOT ON SWAP CHAIN
CALL SWAPIN
AST_UNLOCK (ASTE #) & AST_UNLOCK MASK=AST_UNLOCK_FLAG
N
Y
SEGMENT NOT ON SWAP CHAIN
FIND SEGMENT ON SWAP CHAIN
INDEX = Ø
B
A
IB-50,283


B
A
NEXT = AST_SWAP_CHAIN (INDEX)
INDEX=NEXT
N
NEXT= ASTE #
Y
REMOVE FROM SWAP CHAIN
AST_SWAP_CHAIN (INDEX) = AST_SWAP_CHAIN (ASTE #)
AST_DES_COUNT_(ASTE)=0
AST_UNLOCK (ASTE #) = AST_UNLOCK (ASTE #) & AST_LOCK_MASK
DETERMINE THE MODE OF ACCESS PERMITTED
AST_TYPE (ASTE #) & AST_TYPE_MASK= AST_TYPE_DIRECTORY
Y
MODE =0
N
AST_WAL (ASTE #) & PS_PROCESS_MASK=Ø
N
MODE = SDR_WRITE_ACCESS
Y
MODE = SDR-READ ACCESS
SET PRIORITY LEVEL HIGH
CALL LSD
DESCRIPTORS LOADED IN
C
1B-50,284

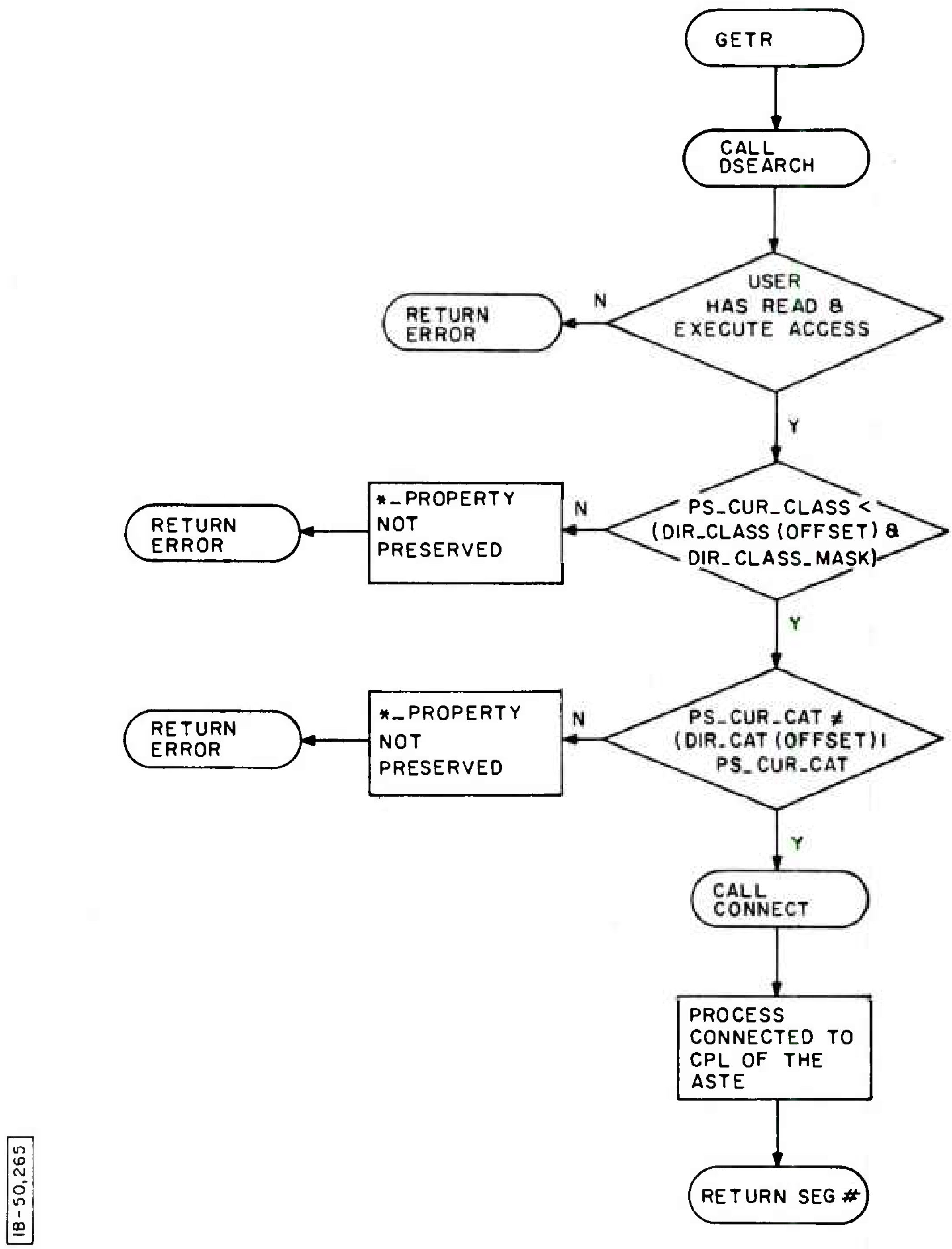

C
PS_CURRENT_
PROCESS=
THE_CURRENT_PROCESS
N
Y
LOAD HARDWARE REGISTERS
SDR (P_REG #)= PS_SDR (REG #)
SAR (P_REG #)= PS_SAR (REG #)
SET PRIORITY LEVEL LOW
INCREMENT DESCRIPTORS
AST_DES_COUNT (ASTE #)=
AST_DES_COUNT (ASTE #)+1
AST_CPL (ASTE #) &
WIRED_DOWN MASK =
WIRED_DOWN
Y
SEGMENT IS NOT IN
USERS MEMORY SPACE
N
ADJUST USERS MEMORY QUOTA
PS_MEM_QUOTA= PS_MEM
QUOTA MINUS AST_SIZE (ASTE #)
RETURN OK
IB-50,285

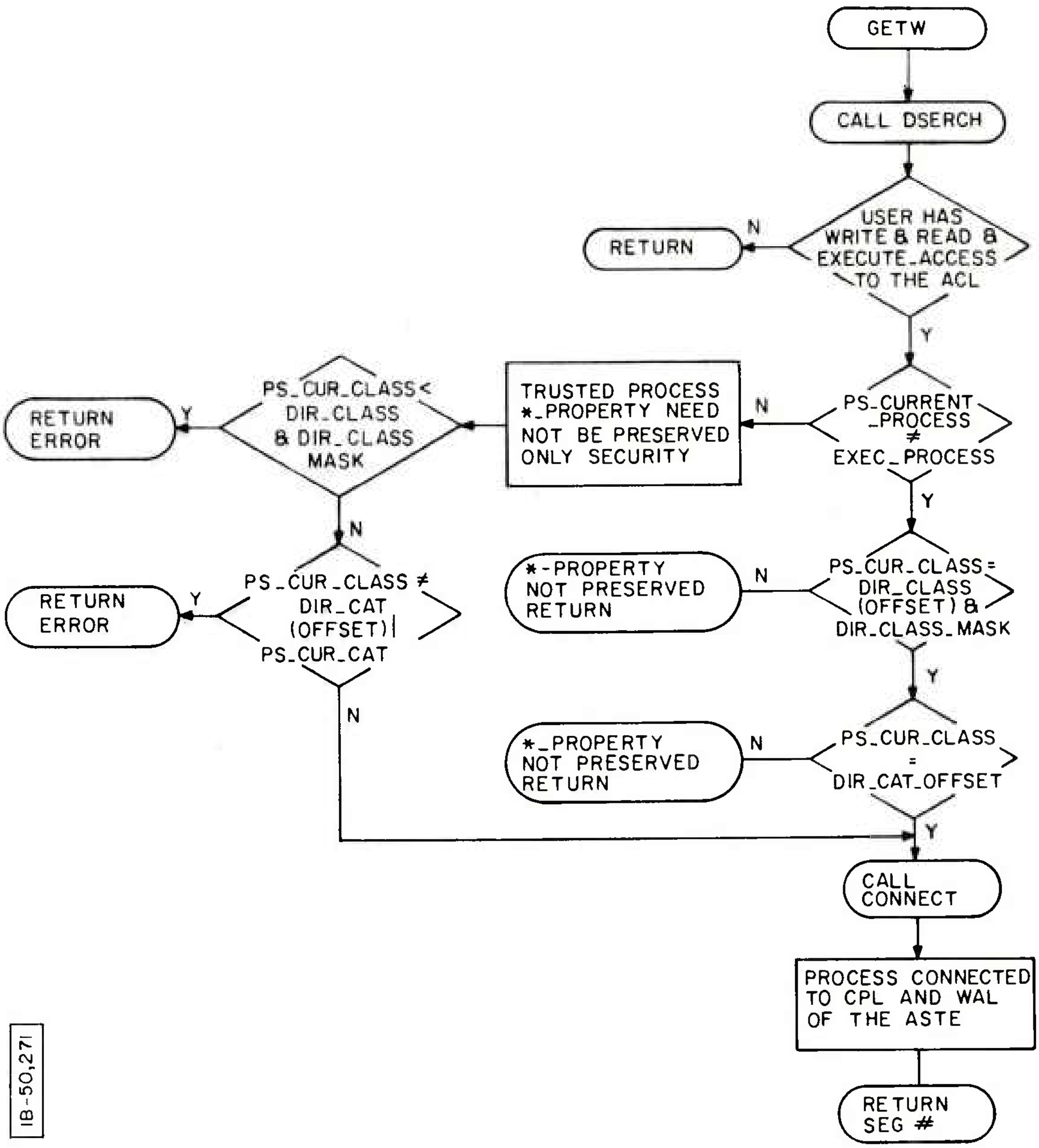

GATE
TRAP
KERNEL_ENTRY
GENERAL REGISTERS SAVED ON STACK
PREVIOUS MODE = SUPERVISOR
N
Y
KSDR3 = SDRO
KSAR3 = SARO
CALL PCHECK
KSDR3 = SDRO
KSAR3 = SARO
PLACE RETURN CODE ON SUPERVISOR STACK
KRC = RC
KERNEL_EXIT
GENERAL REGISTERS RESTORED
RETURN

1B-50,306

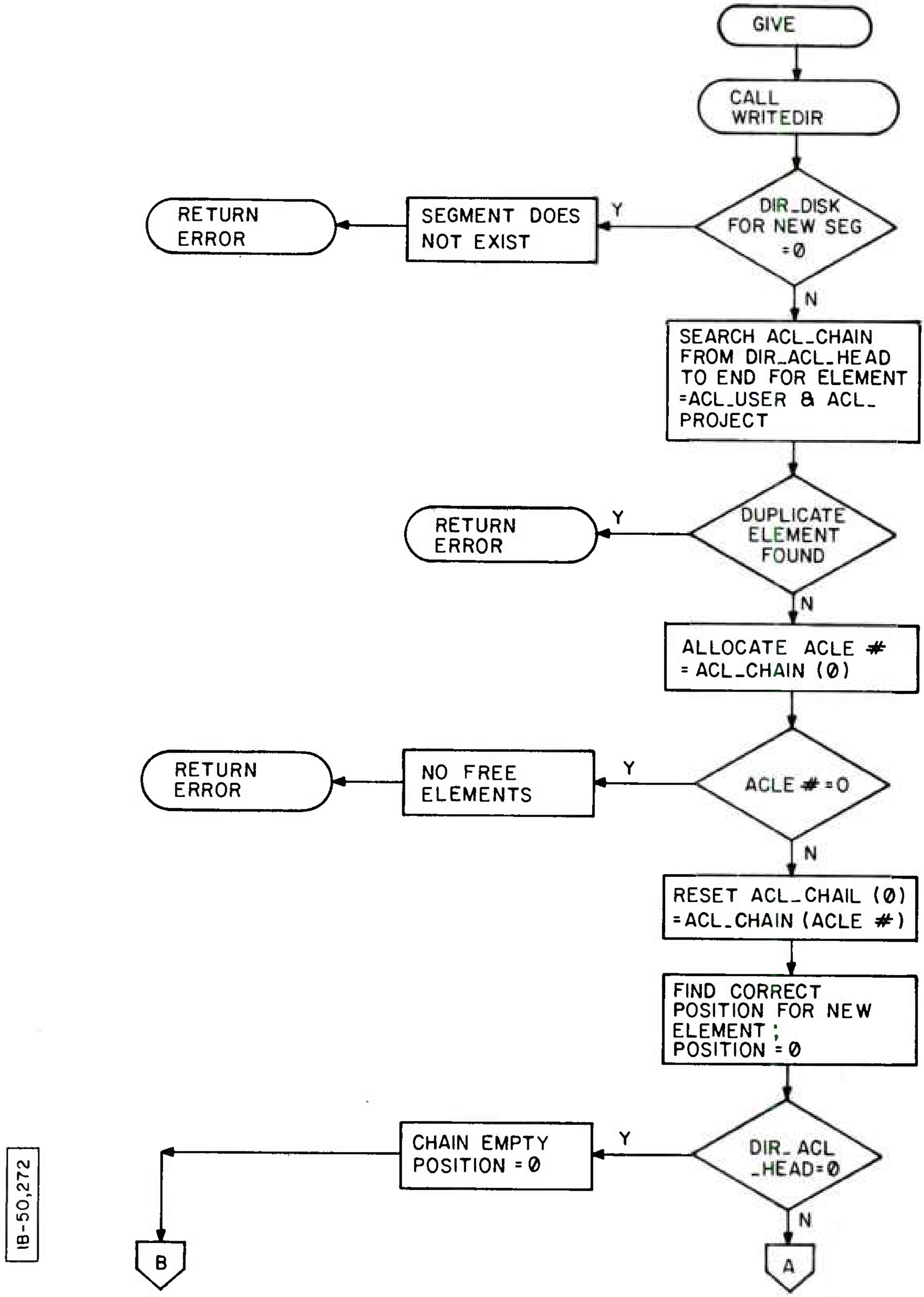

GETDIR
AST_ADR (ASTE#) =Ø ?
Y
N
SEGMENT NOT IN MAIN MEMORY
CALL SWAP N
SEGMENT SWAPPED IN
CALL LSD
SEG DESCRIPTORS IN DIR_KSR_ADR ACCESS SDR_WRITE_ACCESS
RETURN

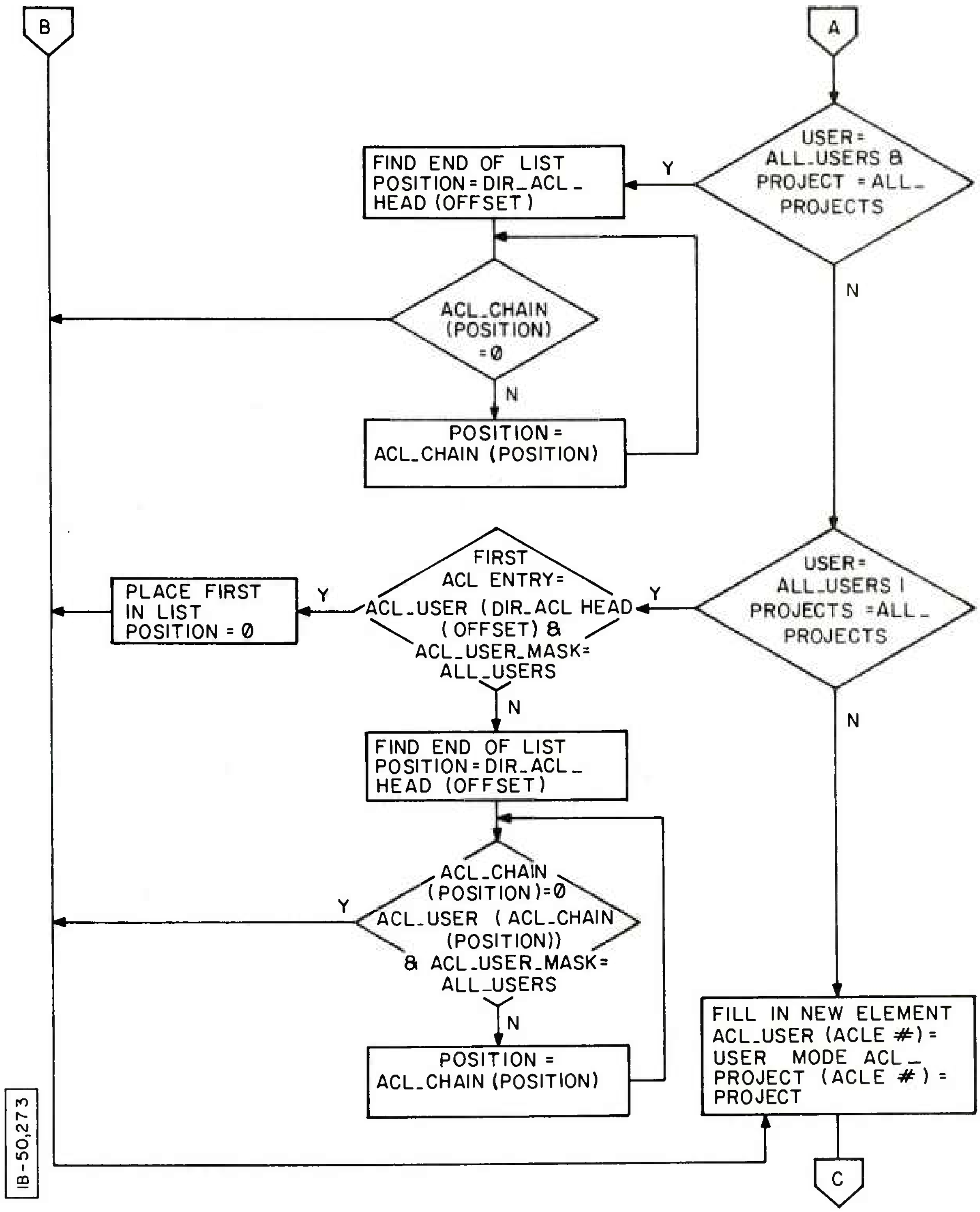
IB-50,264

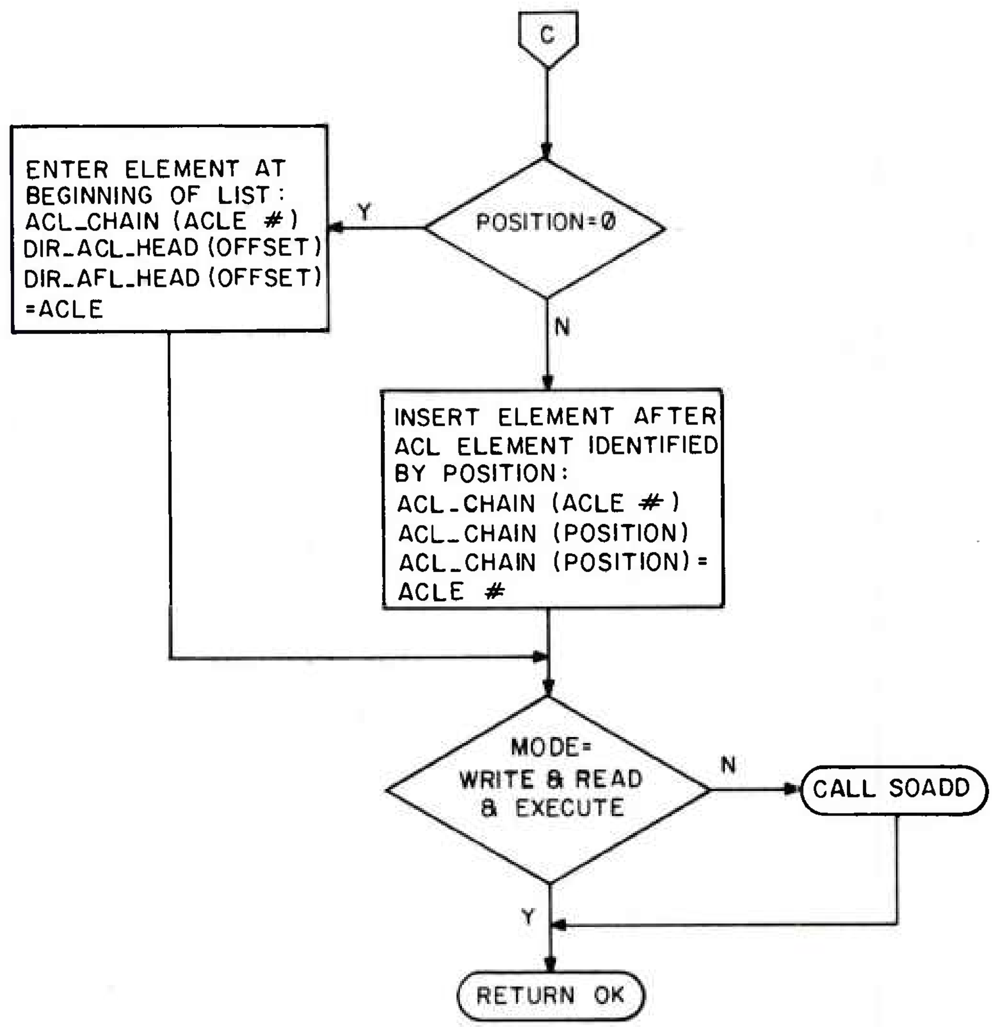

GETR
CALL DSEARCH
USER HAS READ & EXECUTE ACCESS
N
RETURN ERROR
Y
PS_CUR_CLASS < (DIR_CLASS (OFFSET) & DIR_CLASS_MASK)
N
*_PROPERTY NOT PRESERVED
RETURN ERROR
Y
PS_CUR_CAT ≠ (DIR_CAT (OFFSET) | PS_CUR_CAT
N
*_PROPERTY NOT PRESERVED
RETURN ERROR
Y
CALL CONNECT
PROCESS CONNECTED TO CPL OF THE ASTE
RETURN SEG #
IB-50,265


GETW
CALL DSERCH
USER HAS WRITE & READ & EXECUTE_ACCESS TO THE ACL
N
RETURN
Y
PS_CURRENT_PROCESS ≠ EXEC_PROCESS
N
TRUSTED PROCESS *_PROPERTY NEED NOT BE PRESERVED ONLY SECURITY
PS_CUR_CLASS < DIR_CLASS & DIR_CLASS MASK
Y
RETURN ERROR
N
PS_CUR_CLASS ≠ DIR_CAT (OFFSET) \| PS_CUR_CAT
Y
RETURN ERROR
N
Y
PS_CUR_CLASS = DIR_CLASS (OFFSET) & DIR_CLASS_MASK
N
*-PROPERTY NOT PRESERVED RETURN
Y
PS_CUR_CLASS = DIR_CAT_OFFSET
N
*_PROPERTY NOT PRESERVED RETURN
Y
CALL CONNECT
PROCESS CONNECTED TO CPL AND WAL OF THE ASTE
RETURN SEG #

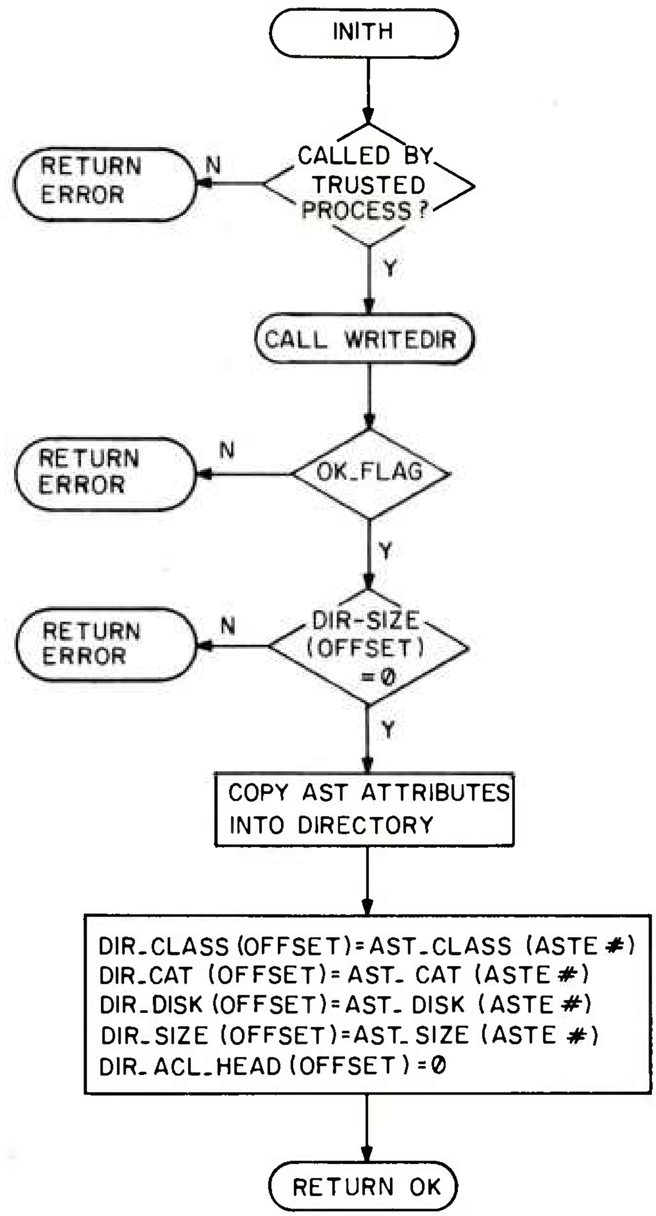
IB-50,271


GIVE
CALL WRITEDIR
DIR_DISK FOR NEW SEG =Ø
Y
SEGMENT DOES NOT EXIST
RETURN ERROR
N
SEARCH ACL_CHAIN FROM DIR_ACL_HEAD TO END FOR ELEMENT =ACL_USER & ACL_PROJECT
DUPLICATE ELEMENT FOUND
Y
RETURN ERROR
N
ALLOCATE ACLE # = ACL_CHAIN (Ø)
ACLE # =0
Y
NO FREE ELEMENTS
RETURN ERROR
N
RESET ACL_CHAIL (Ø) =ACL_CHAIN (ACLE #)
FIND CORRECT POSITION FOR NEW ELEMENT ; POSITION = Ø
DIR_ACL_HEAD=Ø
Y
CHAIN EMPTY POSITION = Ø
B
N
A
IB-50,272

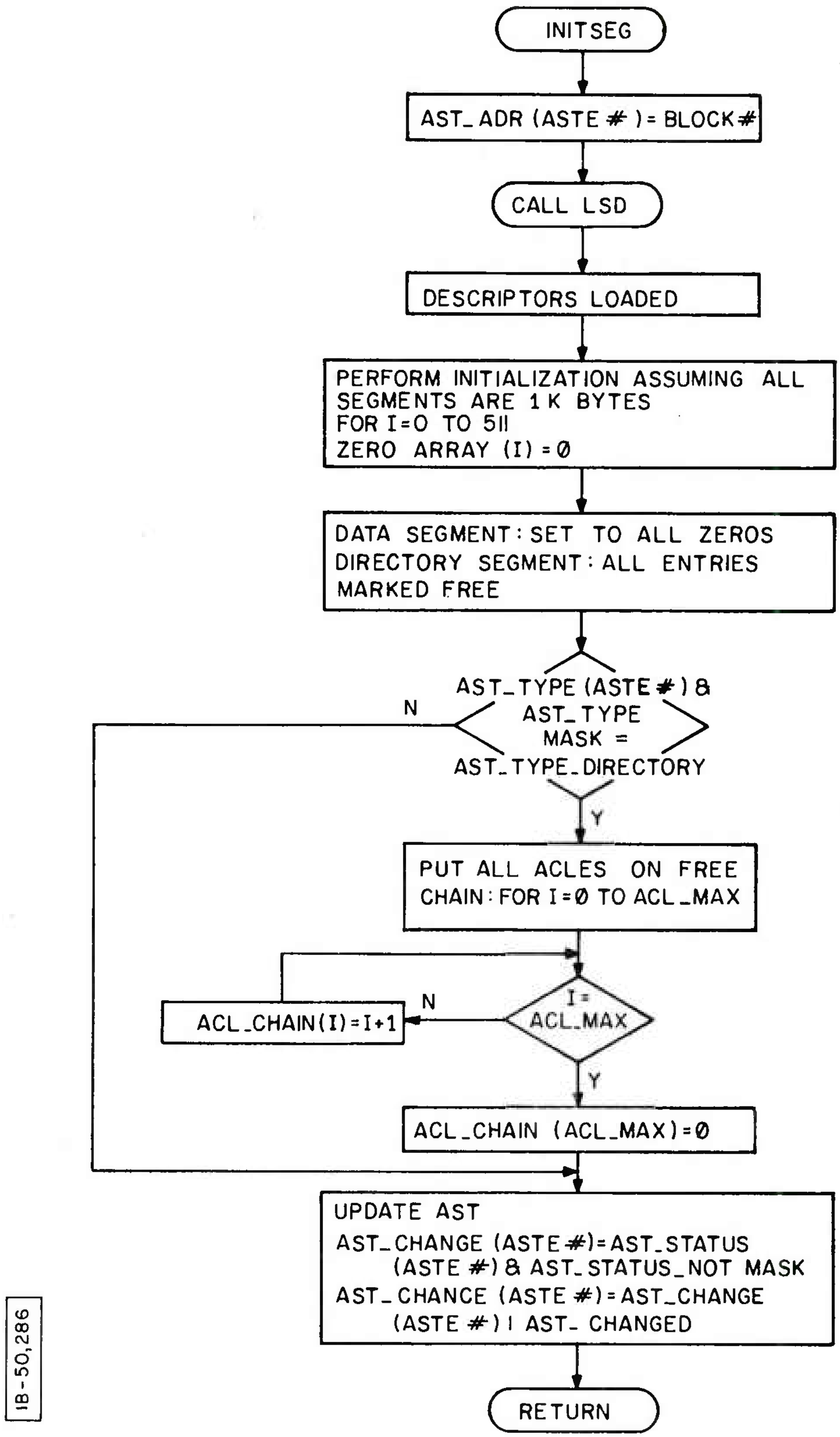

B
A
USER = ALL_USERS & PROJECT = ALL_PROJECTS
Y
FIND END OF LIST POSITION = DIR_ACL_HEAD (OFFSET)
ACL_CHAIN (POSITION) = Ø
N
POSITION = ACL_CHAIN (POSITION)
N
USER = ALL_USERS | PROJECTS = ALL_PROJECTS
Y
FIRST ACL ENTRY = ACL_USER (DIR_ACL HEAD (OFFSET) & ACL_USER_MASK = ALL_USERS
Y
PLACE FIRST IN LIST POSITION = Ø
N
FIND END OF LIST POSITION = DIR_ACL_HEAD (OFFSET)
ACL_CHAIN (POSITION) = Ø ACL_USER (ACL_CHAIN (POSITION)) & ACL_USER_MASK = ALL_USERS
Y
N
POSITION = ACL_CHAIN (POSITION)
N
FILL IN NEW ELEMENT ACL_USER (ACLE #) = USER MODE ACL_PROJECT (ACLE #) = PROJECT
C
IB-50,273


C
POSITION=Ø
Y
N
ENTER ELEMENT AT BEGINNING OF LIST: ACL_CHAIN (ACLE #) DIR_ACL_HEAD (OFFSET) DIR_AFL_HEAD (OFFSET) =ACLE
INSERT ELEMENT AFTER ACL ELEMENT IDENTIFIED BY POSITION: ACL_CHAIN (ACLE #) ACL_CHAIN (POSITION) ACL_CHAIN (POSITION) = ACLE #
MODE= WRITE & READ & EXECUTE
N
CALL SOADD
Y
RETURN OK

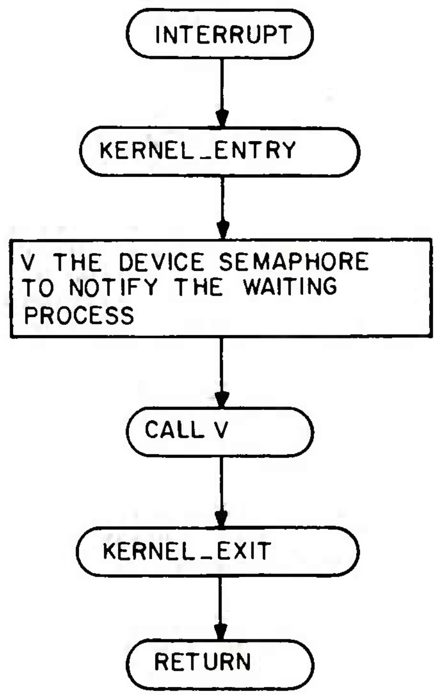
IB-50,274


INITH
CALLED BY TRUSTED PROCESS ?
N
RETURN ERROR
Y
CALL WRITEDIR
OK_FLAG
N
RETURN ERROR
Y
DIR-SIZE (OFFSET) =Ø
N
RETURN ERROR
Y
COPY AST ATTRIBUTES INTO DIRECTORY
DIR_CLASS (OFFSET)=AST_CLASS (ASTE #)
DIR_CAT (OFFSET)=AST_CAT (ASTE #)
DIR_DISK (OFFSET)=AST_DISK (ASTE #)
DIR_SIZE (OFFSET)=AST_SIZE (ASTE #)
DIR_ACL_HEAD (OFFSET)=Ø
RETURN OK

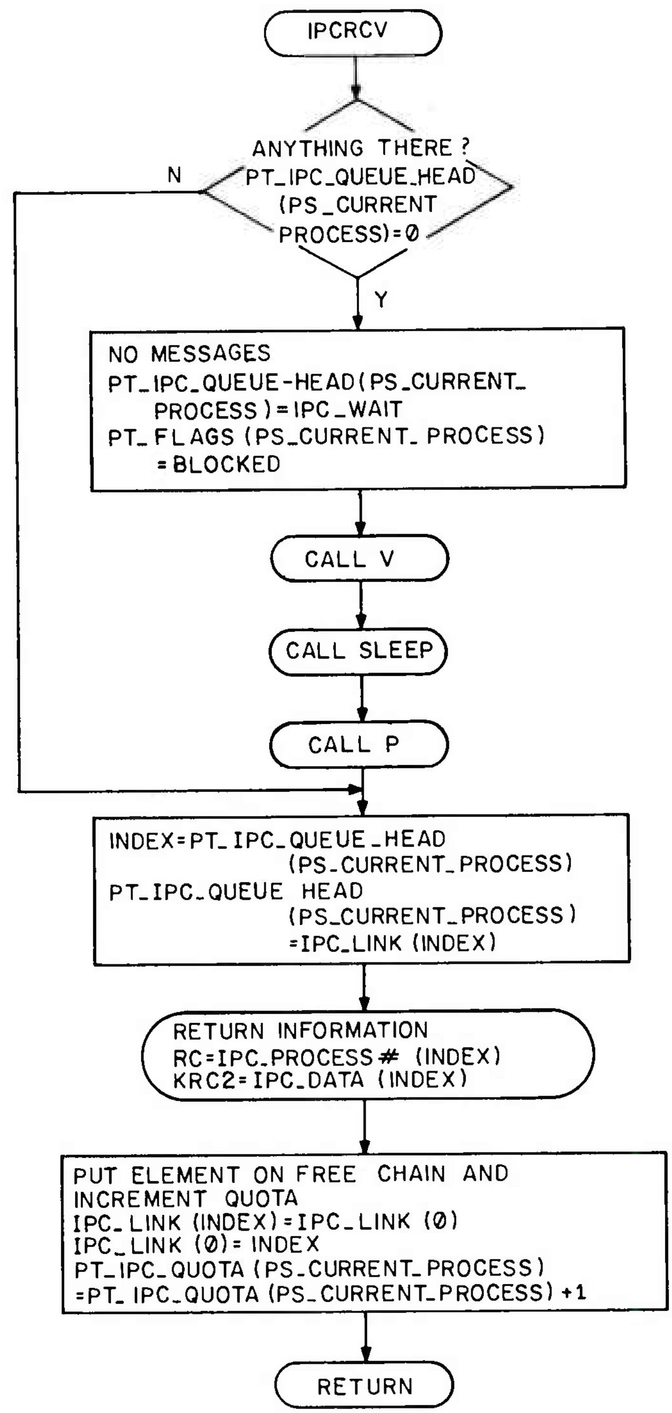
IB-50,305


INITSEG
AST_ADR (ASTE #) = BLOCK#
CALL LSD
DESCRIPTORS LOADED
PERFORM INITIALIZATION ASSUMING ALL SEGMENTS ARE 1 K BYTES
FOR I=0 TO 511
ZERO ARRAY (I) = Ø
DATA SEGMENT: SET TO ALL ZEROS
DIRECTORY SEGMENT: ALL ENTRIES MARKED FREE
AST_TYPE (ASTE #) & AST_TYPE MASK = AST_TYPE_DIRECTORY
N
Y
PUT ALL ACLES ON FREE CHAIN: FOR I = Ø TO ACL_MAX
ACL_CHAIN(I) = I+1
N
I = ACL_MAX
Y
ACL_CHAIN (ACL_MAX) = Ø
UPDATE AST
AST_CHANGE (ASTE #) = AST_STATUS (ASTE #) & AST_STATUS_NOT MASK
AST_CHANCE (ASTE #) = AST_CHANGE (ASTE #) I AST_CHANGED
RETURN
IB-50,286

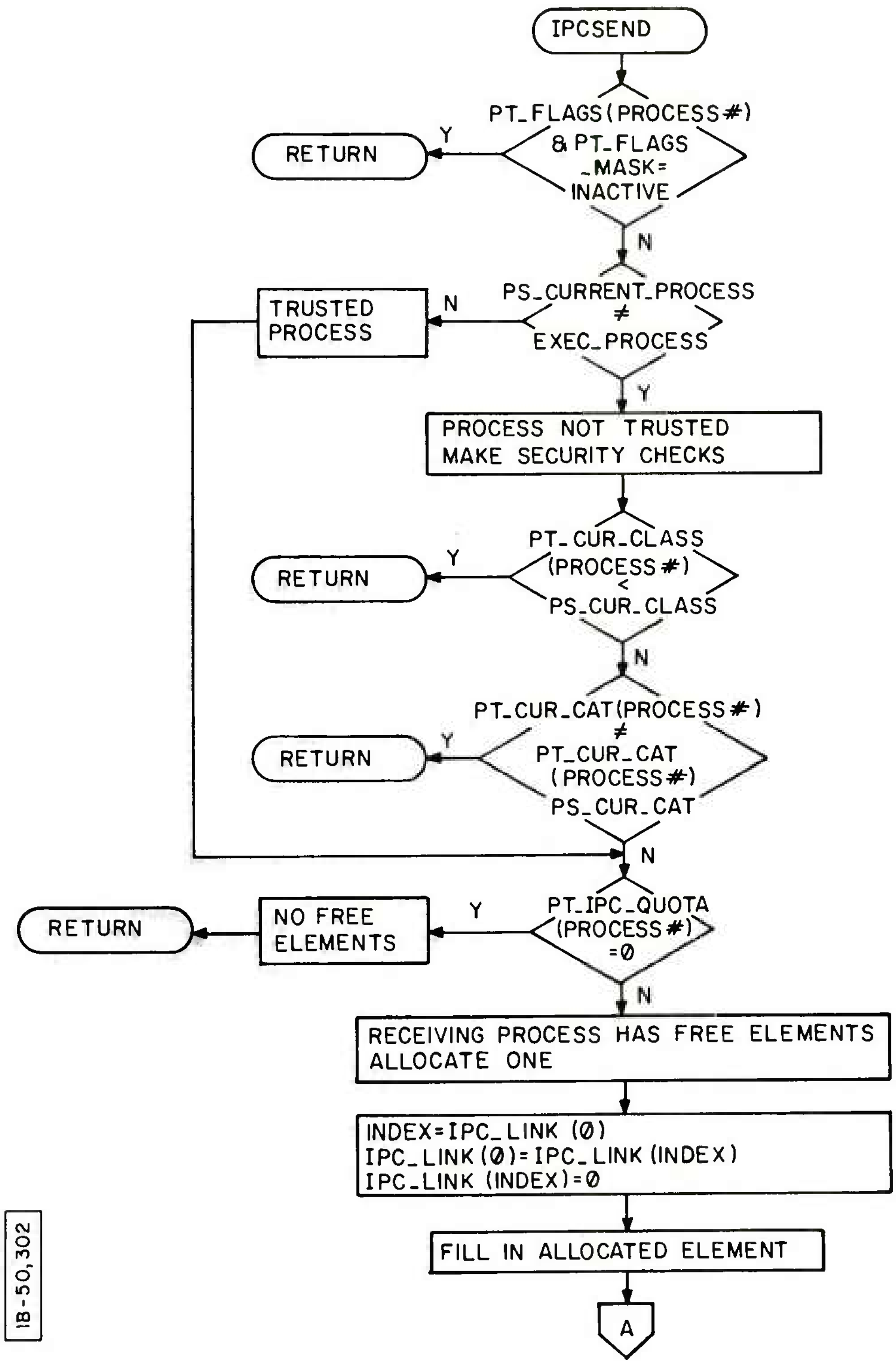
NOTE: ALL INTERRUPT HANDLERS LISTED IN GATE WORK
THE SAME WAY.

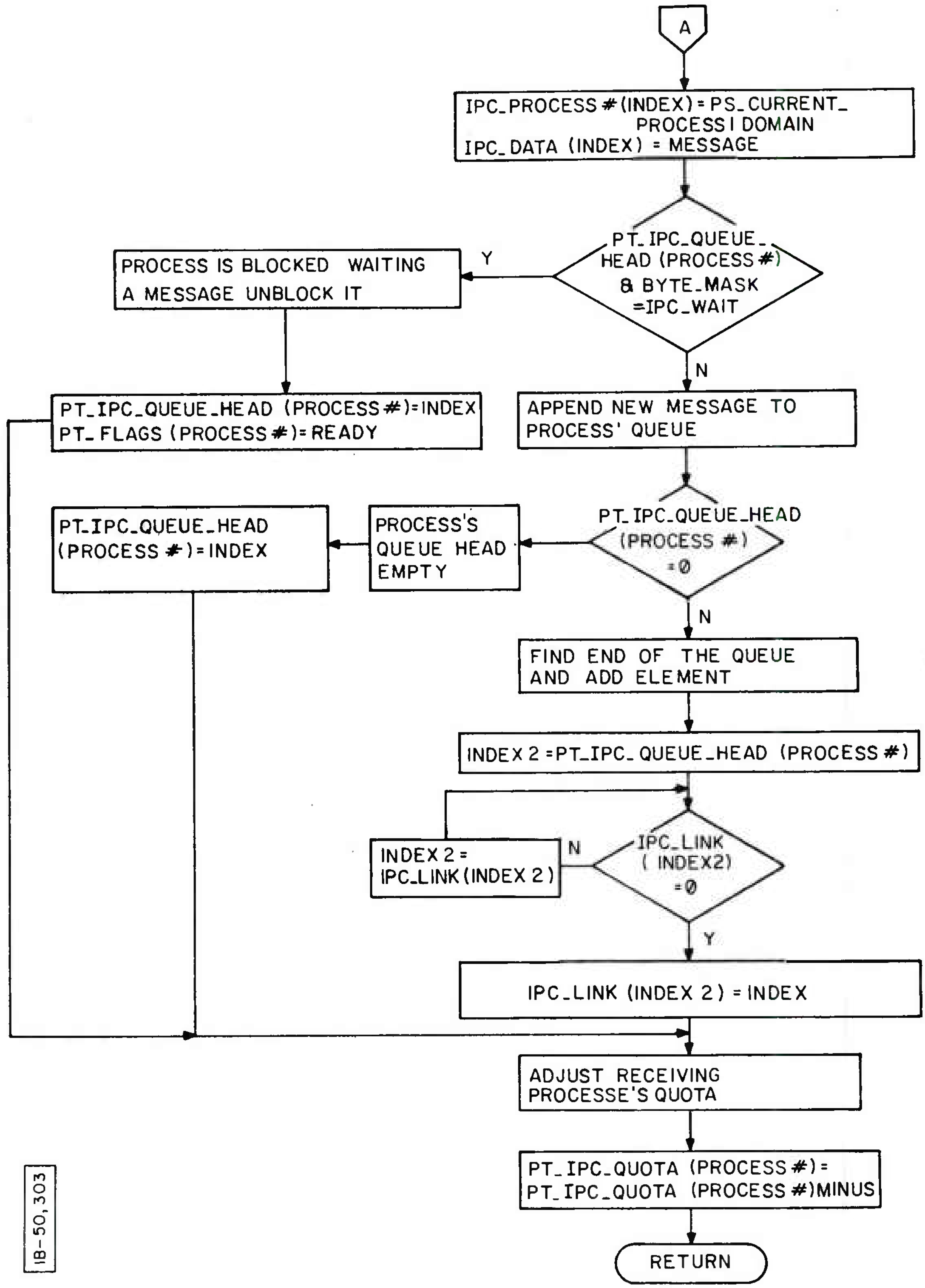


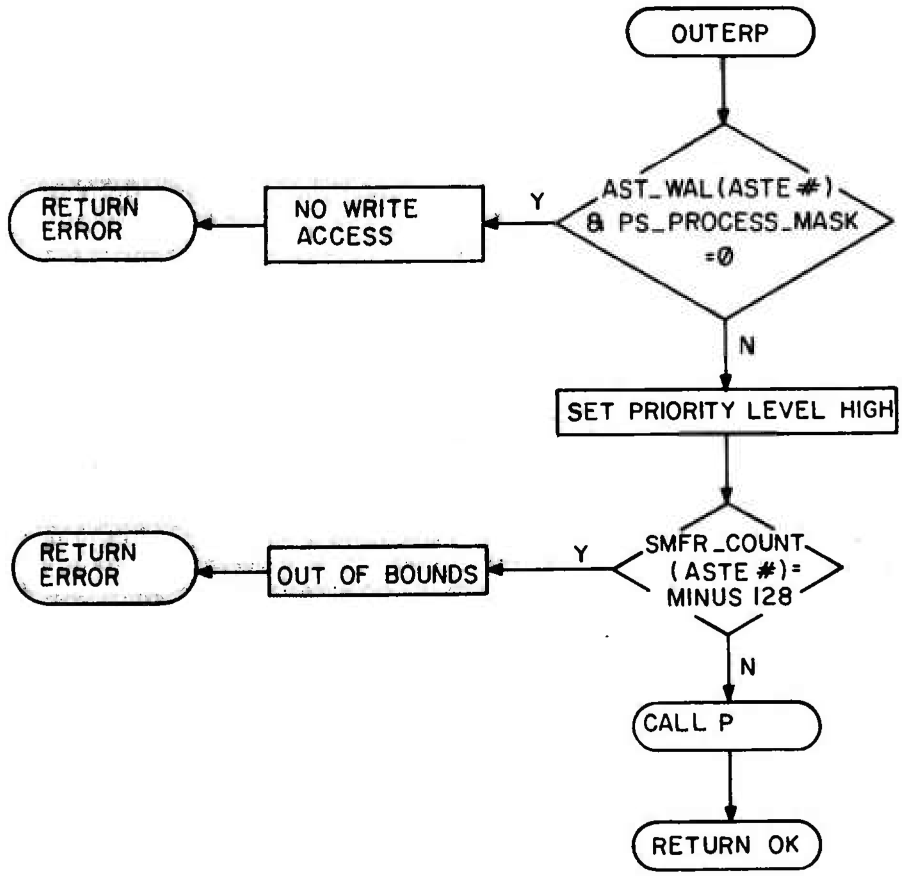
IB-50,304


IPCRCV
ANYTHING THERE ?
PT_IPC_QUEUE_HEAD
(PS_CURRENT
PROCESS)=Ø
N
Y
NO MESSAGES
PT_IPC_QUEUE-HEAD(PS_CURRENT_
PROCESS)=IPC_WAIT
PT_FLAGS (PS_CURRENT_PROCESS)
=BLOCKED
CALL V
CALL SLEEP
CALL P
INDEX=PT_IPC_QUEUE_HEAD
(PS_CURRENT_PROCESS)
PT_IPC_QUEUE HEAD
(PS_CURRENT_PROCESS)
=IPC_LINK (INDEX)
RETURN INFORMATION
RC=IPC_PROCESS # (INDEX)
KRC2=IPC_DATA (INDEX)
PUT ELEMENT ON FREE CHAIN AND
INCREMENT QUOTA
IPC_LINK (INDEX)=IPC_LINK (Ø)
IPC_LINK (Ø)= INDEX
PT_IPC_QUOTA (PS_CURRENT_PROCESS)
=PT_IPC_QUOTA (PS_CURRENT_PROCESS) +1
RETURN

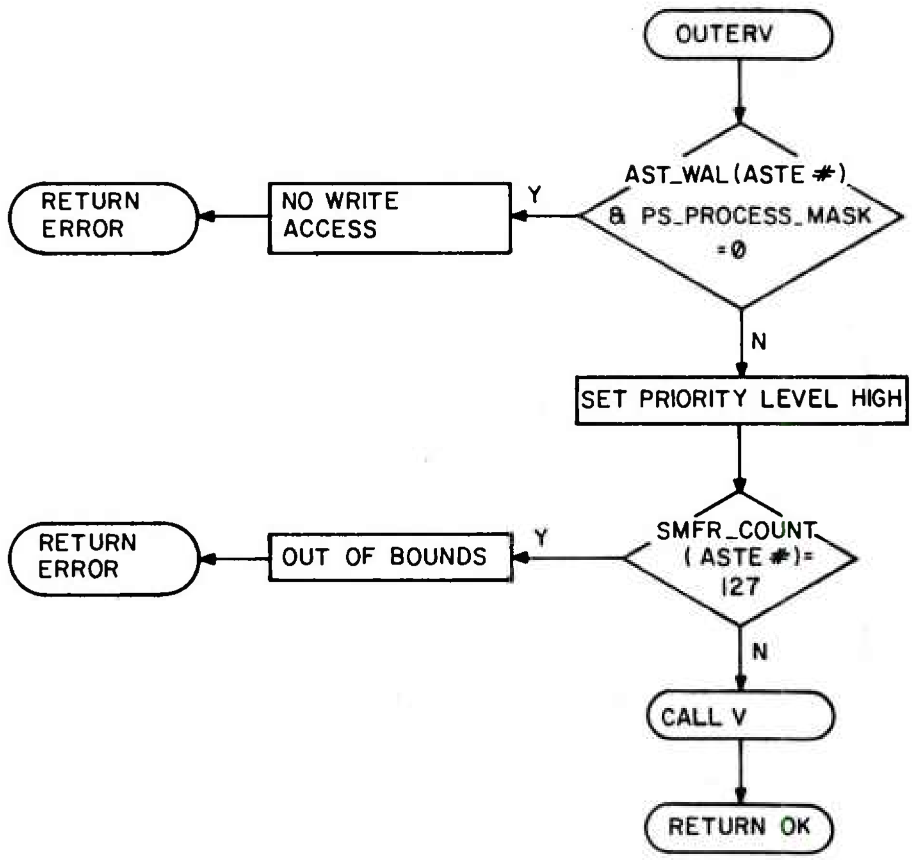
IB-50,313


IPCSEND
PT_FLAGS(PROCESS#) & PT_FLAGS _MASK= INACTIVE
Y
RETURN
N
PS_CURRENT_PROCESS ≠ EXEC_PROCESS
N
TRUSTED PROCESS
Y
PROCESS NOT TRUSTED MAKE SECURITY CHECKS
PT_CUR_CLASS (PROCESS#) < PS_CUR_CLASS
Y
RETURN
N
PT_CUR_CAT(PROCESS#) ≠ PT_CUR_CAT (PROCESS#) PS_CUR_CAT
Y
RETURN
N
PT_IPC_QUOTA (PROCESS#) =Ø
Y
NO FREE ELEMENTS
RETURN
N
RECEIVING PROCESS HAS FREE ELEMENTS ALLOCATE ONE
INDEX=IPC_LINK (Ø)
IPC_LINK (Ø)=IPC_LINK (INDEX)
IPC_LINK (INDEX)=Ø
FILL IN ALLOCATED ELEMENT
A
IB-50,302

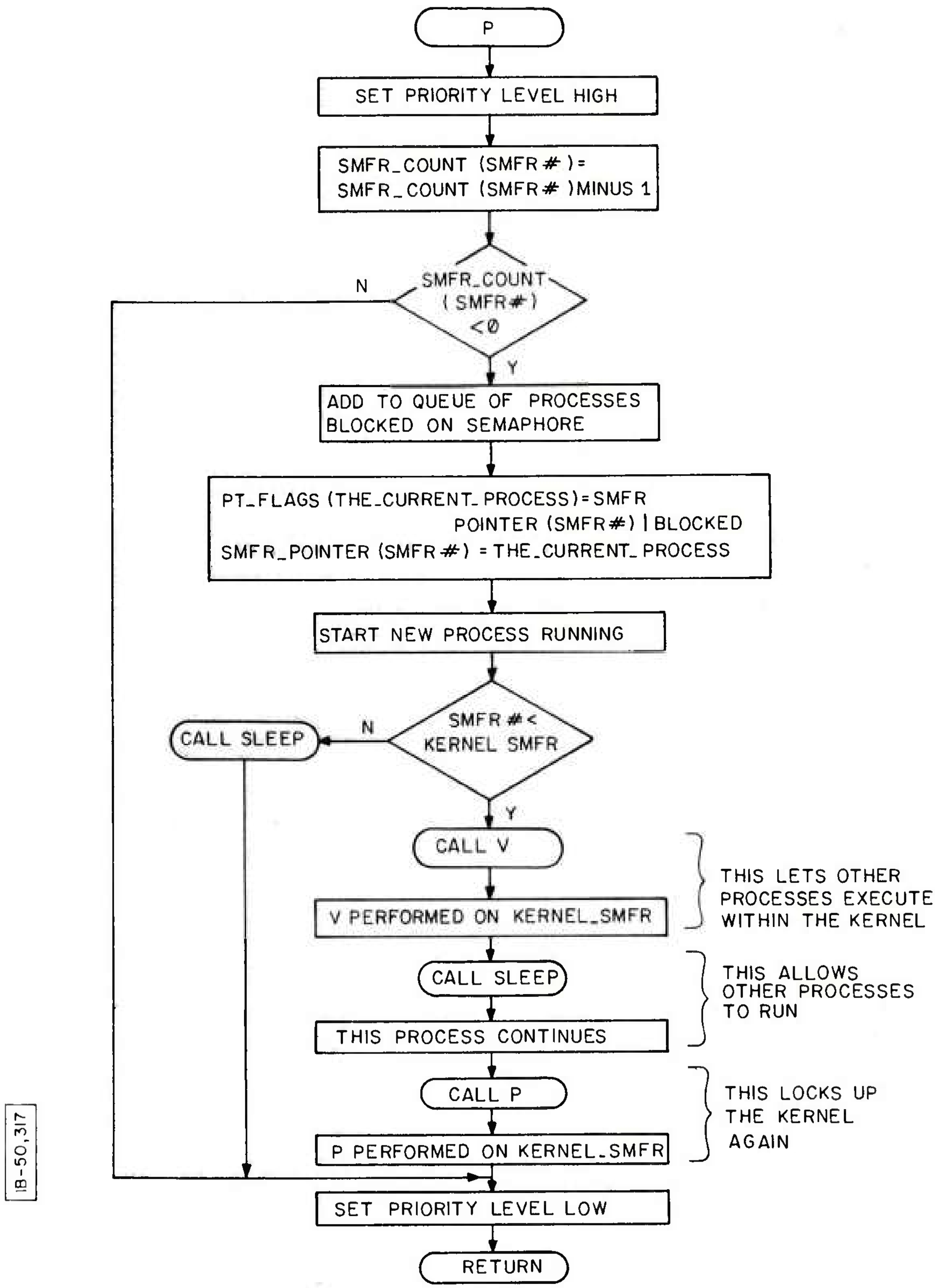

A
IPC_PROCESS # (INDEX) = PS_CURRENT_PROCESS I DOMAIN
IPC_DATA (INDEX) = MESSAGE
PT_IPC_QUEUE_HEAD (PROCESS #) & BYTE_MASK =IPC_WAIT
Y
PROCESS IS BLOCKED WAITING A MESSAGE UNBLOCK IT
PT_IPC_QUEUE_HEAD (PROCESS #)=INDEX
PT_FLAGS (PROCESS #)= READY
N
APPEND NEW MESSAGE TO PROCESS' QUEUE
PT_IPC_QUEUE_HEAD (PROCESS #) =Ø
PROCESS'S QUEUE HEAD EMPTY
PT_IPC_QUEUE_HEAD (PROCESS #) = INDEX
N
FIND END OF THE QUEUE AND ADD ELEMENT
INDEX 2 =PT_IPC_QUEUE_HEAD (PROCESS #)
IPC_LINK (INDEX2) =Ø
N
INDEX 2 = IPC_LINK (INDEX 2)
Y
IPC_LINK (INDEX 2) = INDEX
ADJUST RECEIVING PROCESSE'S QUOTA
PT_IPC_QUOTA (PROCESS #) = PT_IPC_QUOTA (PROCESS #)MINUS
RETURN
IB-50,303

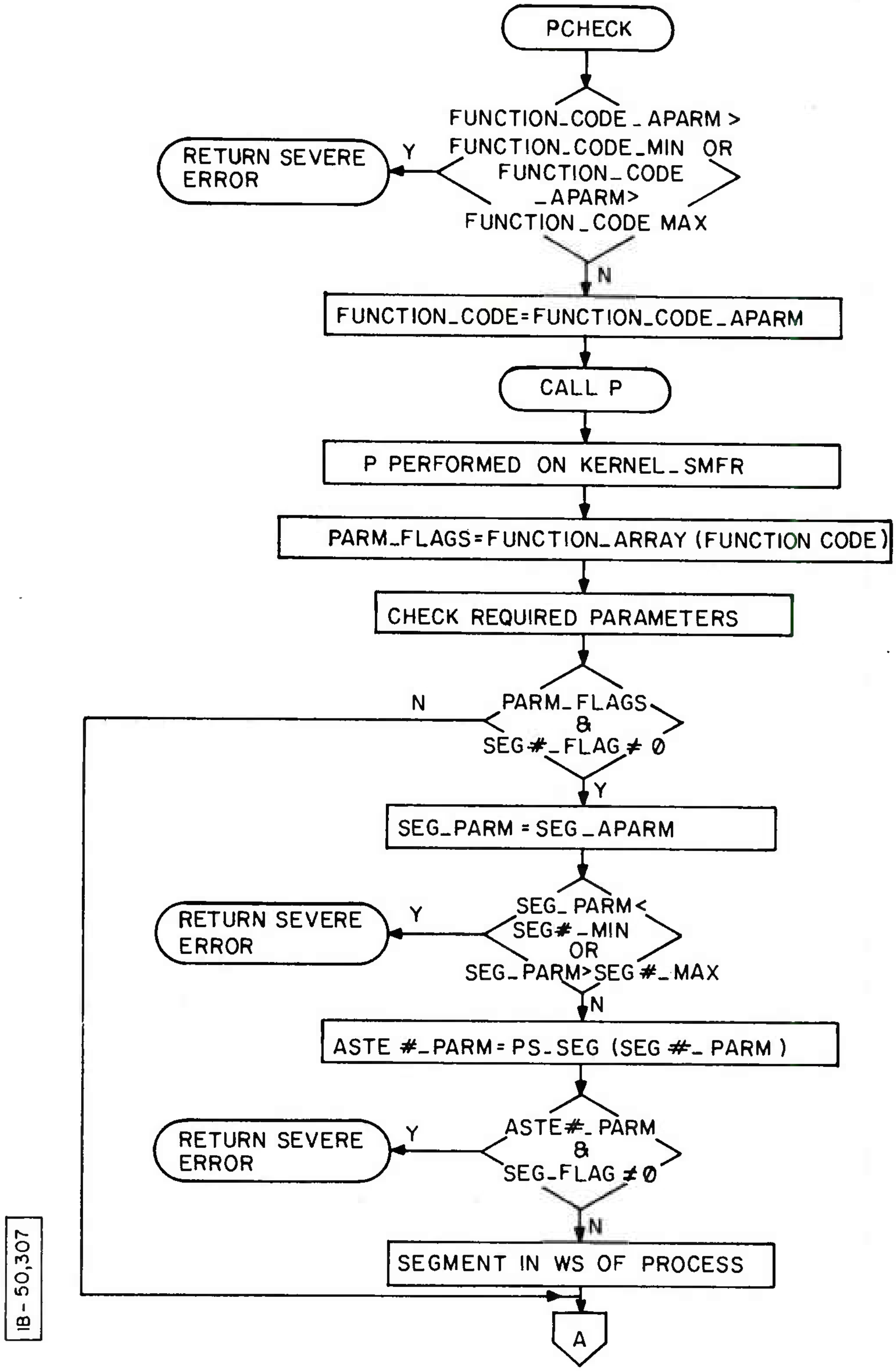

OUTERP
AST_WAL(ASTE #)
& PS_PROCESS_MASK
=Ø
Y
N
NO WRITE
ACCESS
RETURN
ERROR
SET PRIORITY LEVEL HIGH
SMFR_COUNT
(ASTE #)=
MINUS 128
Y
N
OUT OF BOUNDS
RETURN
ERROR
CALL P
RETURN OK

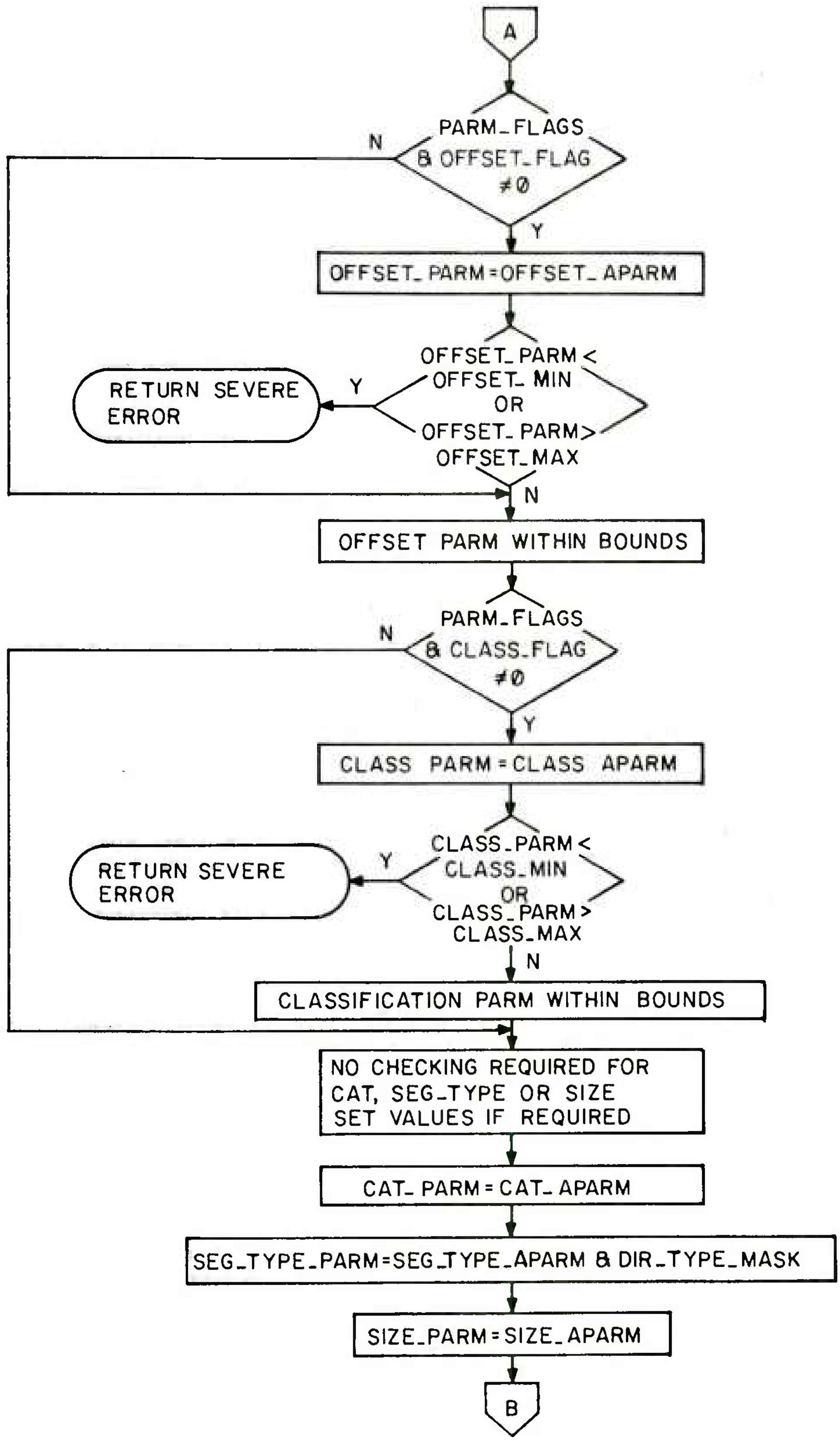
IB-50,298


OUTERV
AST_WAL (ASTE #)
& PS_PROCESS_MASK
=Ø
Y
NO WRITE
ACCESS
RETURN
ERROR
N
SET PRIORITY LEVEL HIGH
SMFR_COUNT
(ASTE #)=
127
Y
OUT OF BOUNDS
RETURN
ERROR
N
CALL V
RETURN OK

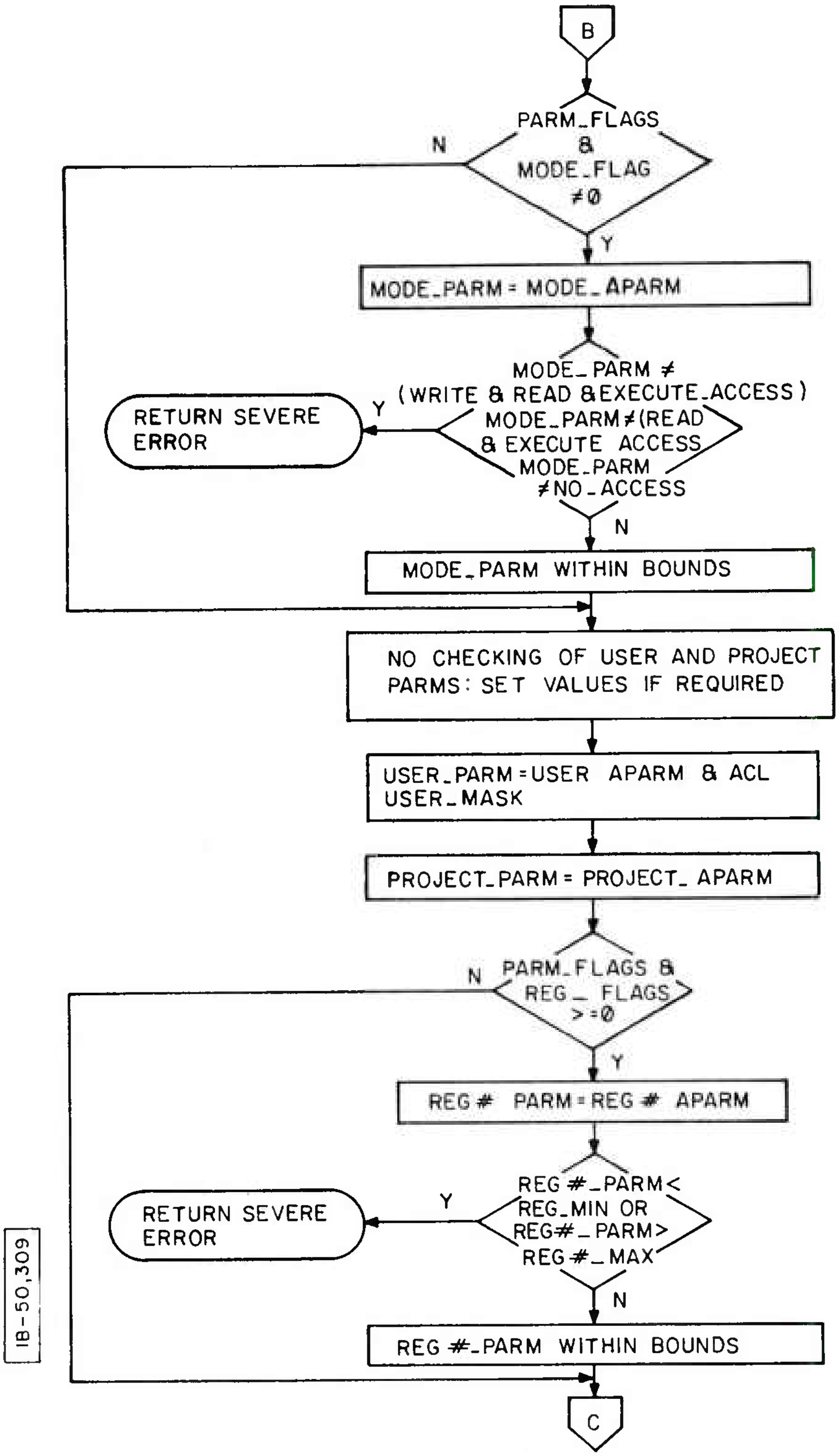
IB-50,299

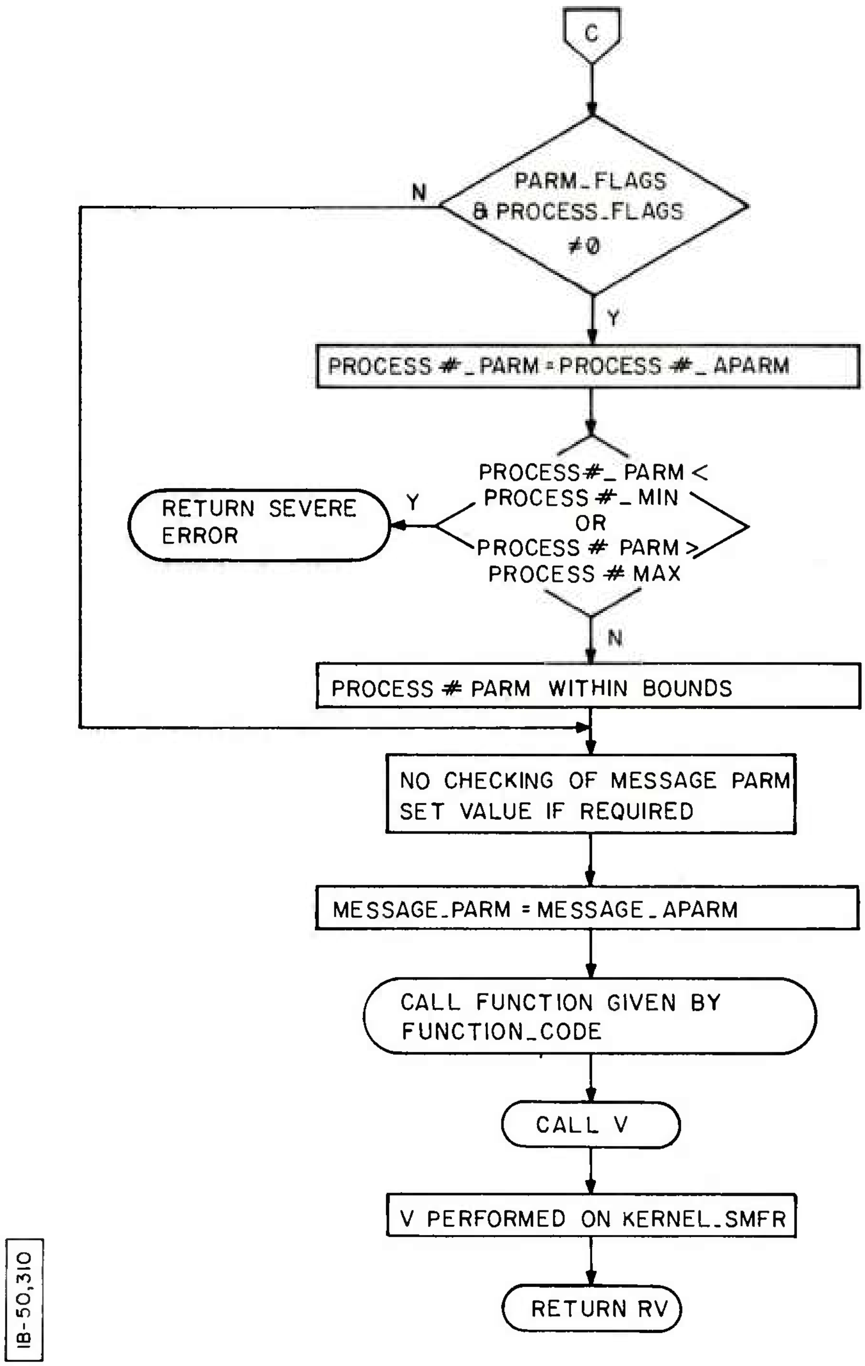

P
SET PRIORITY LEVEL HIGH
SMFR_COUNT (SMFR#)=
SMFR_COUNT (SMFR#)MINUS 1
SMFR_COUNT (SMFR#) <0
N
Y
ADD TO QUEUE OF PROCESSES BLOCKED ON SEMAPHORE
PT_FLAGS (THE_CURRENT_PROCESS)=SMFR POINTER (SMFR#) | BLOCKED
SMFR_POINTER (SMFR#) = THE_CURRENT_PROCESS
START NEW PROCESS RUNNING
SMFR # < KERNEL SMFR
N
CALL SLEEP
Y
CALL V
V PERFORMED ON KERNEL_SMFR
THIS LETS OTHER PROCESSES EXECUTE WITHIN THE KERNEL
CALL SLEEP
THIS PROCESS CONTINUES
THIS ALLOWS OTHER PROCESSES TO RUN
CALL P
P PERFORMED ON KERNEL_SMFR
THIS LOCKS UP THE KERNEL AGAIN
SET PRIORITY LEVEL LOW
RETURN
1B-50,317

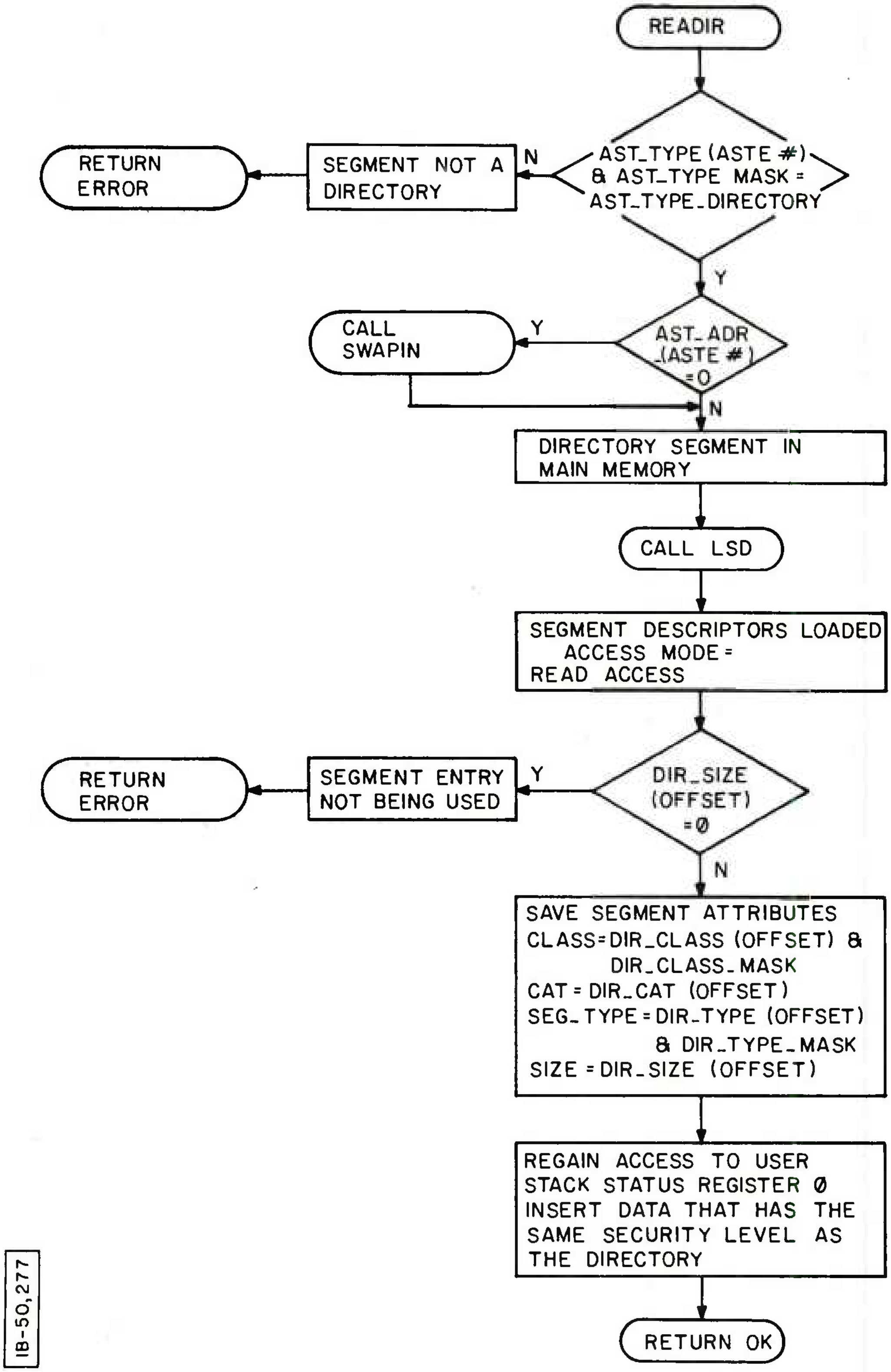

PCHECK
FUNCTION_CODE_APARM > FUNCTION_CODE_MIN OR FUNCTION_CODE_APARM> FUNCTION_CODE MAX
Y
RETURN SEVERE ERROR
N
FUNCTION_CODE=FUNCTION_CODE_APARM
CALL P
P PERFORMED ON KERNEL_SMFR
PARM_FLAGS=FUNCTION_ARRAY (FUNCTION CODE)
CHECK REQUIRED PARAMETERS
PARM_FLAGS & SEG#_FLAG ≠ 0
N
Y
SEG_PARM = SEG_APARM
SEG_PARM< SEG#_MIN OR SEG_PARM>SEG#_MAX
Y
RETURN SEVERE ERROR
N
ASTE #_PARM = PS_SEG (SEG #_PARM)
ASTE#_PARM & SEG_FLAG ≠ 0
Y
RETURN SEVERE ERROR
N
SEGMENT IN WS OF PROCESS
A
IB-50,307

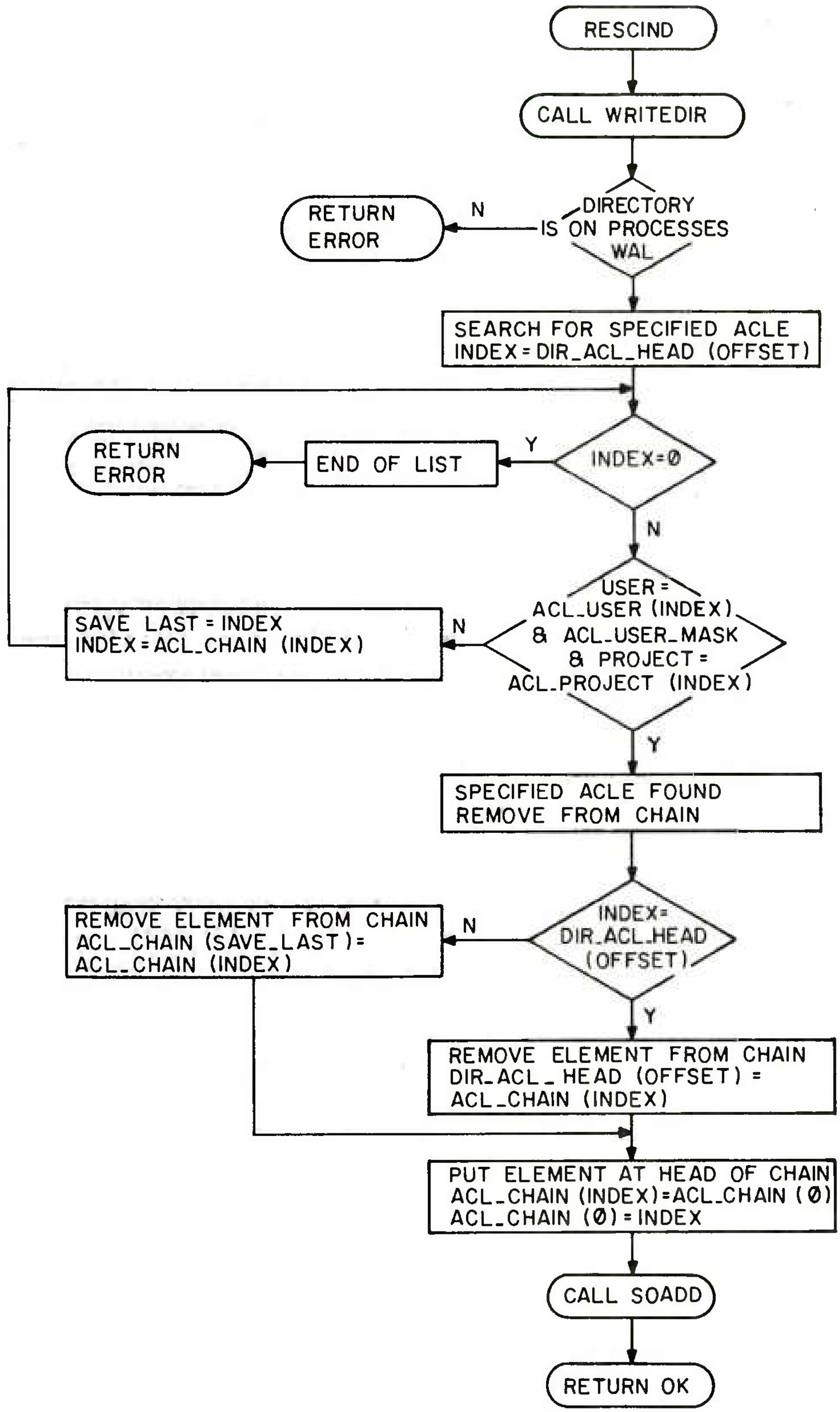

A
PARM_FLAGS & OFFSET_FLAG ≠0
N
Y
OFFSET_PARM=OFFSET_APARM
OFFSET_PARM < OFFSET_MIN OR OFFSET_PARM > OFFSET_MAX
Y
RETURN SEVERE ERROR
N
OFFSET PARM WITHIN BOUNDS
PARM_FLAGS & CLASS_FLAG ≠0
N
Y
CLASS PARM = CLASS APARM
CLASS_PARM < CLASS_MIN OR CLASS_PARM > CLASS_MAX
Y
RETURN SEVERE ERROR
N
CLASSIFICATION PARM WITHIN BOUNDS
NO CHECKING REQUIRED FOR CAT, SEG_TYPE OR SIZE SET VALUES IF REQUIRED
CAT_PARM = CAT_APARM
SEG_TYPE_PARM=SEG_TYPE_APARM & DIR_TYPE_MASK
SIZE_PARM = SIZE_APARM
B

IB-50,308


B
PARM_FLAGS & MODE_FLAG ≠Ø
N
Y
MODE_PARM = MODE_APARM
MODE_PARM ≠ (WRITE & READ & EXECUTE_ACCESS) MODE_PARM ≠ (READ & EXECUTE ACCESS MODE_PARM ≠ NO_ACCESS
Y
RETURN SEVERE ERROR
N
MODE_PARM WITHIN BOUNDS
NO CHECKING OF USER AND PROJECT PARMS: SET VALUES IF REQUIRED
USER_PARM = USER APARM & ACL USER_MASK
PROJECT_PARM = PROJECT_APARM
PARM_FLAGS & REG_FLAGS >=Ø
N
Y
REG # PARM = REG # APARM
REG #_PARM < REG_MIN OR REG #_PARM > REG #_MAX
Y
RETURN SEVERE ERROR
N
REG #_PARM WITHIN BOUNDS
C
IB-50,309

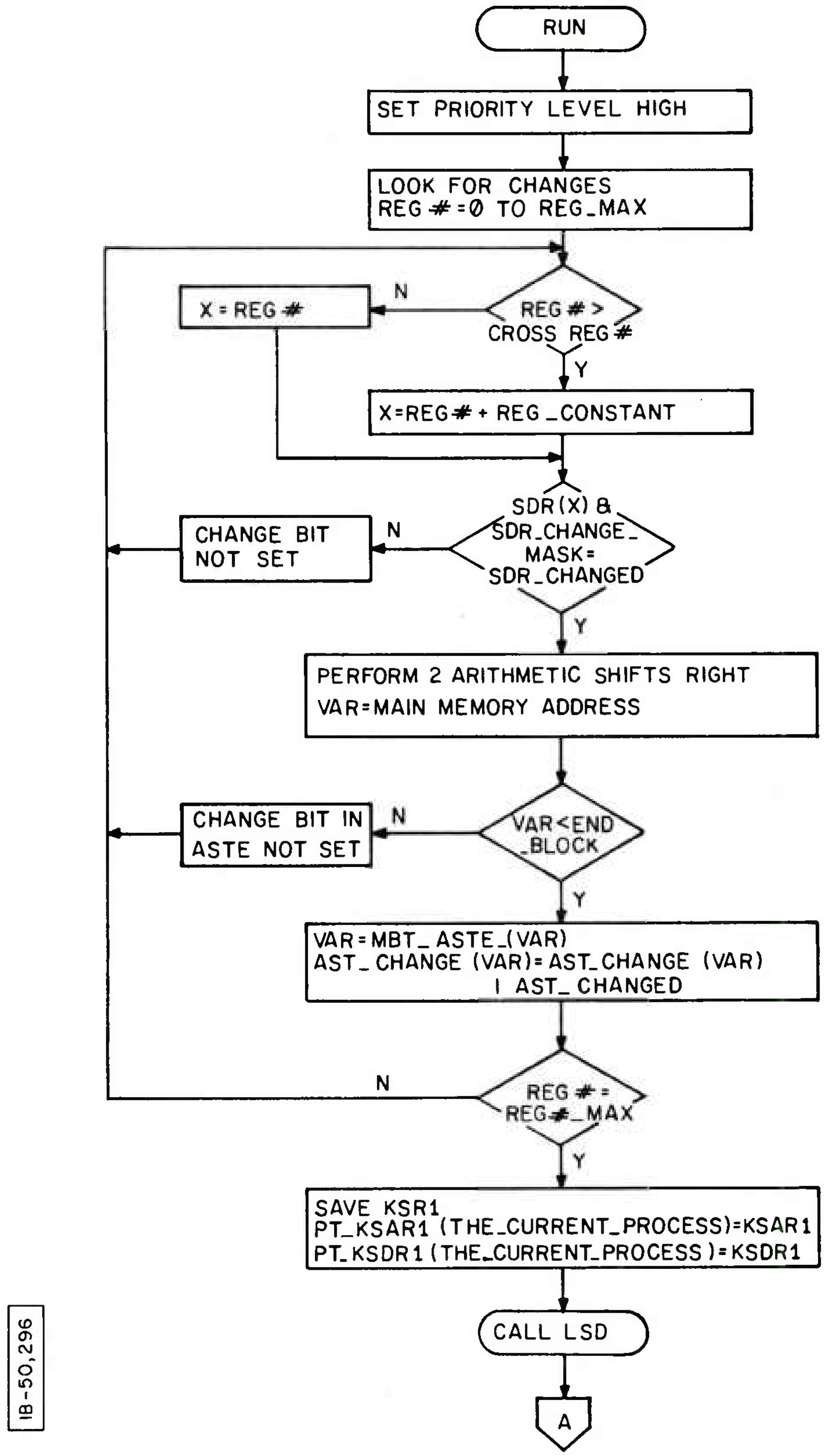

C
PARM_FLAGS & PROCESS_FLAGS ≠Ø
N
Y
PROCESS #_PARM = PROCESS #_APARM
PROCESS#_PARM < PROCESS #_MIN OR PROCESS # PARM > PROCESS # MAX
Y
RETURN SEVERE ERROR
N
PROCESS # PARM WITHIN BOUNDS
NO CHECKING OF MESSAGE PARM SET VALUE IF REQUIRED
MESSAGE_PARM = MESSAGE_APARM
CALL FUNCTION GIVEN BY FUNCTION_CODE
CALL V
V PERFORMED ON KERNEL_SMFR
RETURN RV
IB-50,310

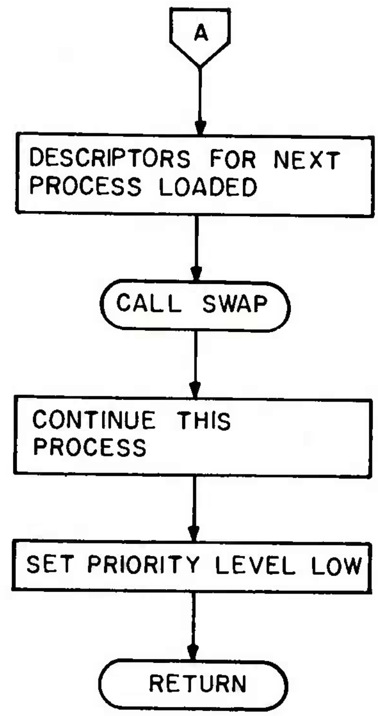

READIR
AST_TYPE (ASTE #)
& AST_TYPE MASK =
AST_TYPE_DIRECTORY
N
SEGMENT NOT A
DIRECTORY
RETURN
ERROR
Y
AST_ADR
(ASTE #)
=0
Y
CALL
SWAPIN
N
DIRECTORY SEGMENT IN
MAIN MEMORY
CALL LSD
SEGMENT DESCRIPTORS LOADED
ACCESS MODE =
READ ACCESS
DIR_SIZE
(OFFSET)
=Ø
Y
SEGMENT ENTRY
NOT BEING USED
RETURN
ERROR
N
SAVE SEGMENT ATTRIBUTES
CLASS=DIR_CLASS (OFFSET) &
DIR_CLASS_MASK
CAT = DIR_CAT (OFFSET)
SEG_TYPE = DIR_TYPE (OFFSET)
& DIR_TYPE_MASK
SIZE = DIR_SIZE (OFFSET)
REGAIN ACCESS TO USER
STACK STATUS REGISTER Ø
INSERT DATA THAT HAS THE
SAME SECURITY LEVEL AS
THE DIRECTORY
RETURN OK
IB-50,277



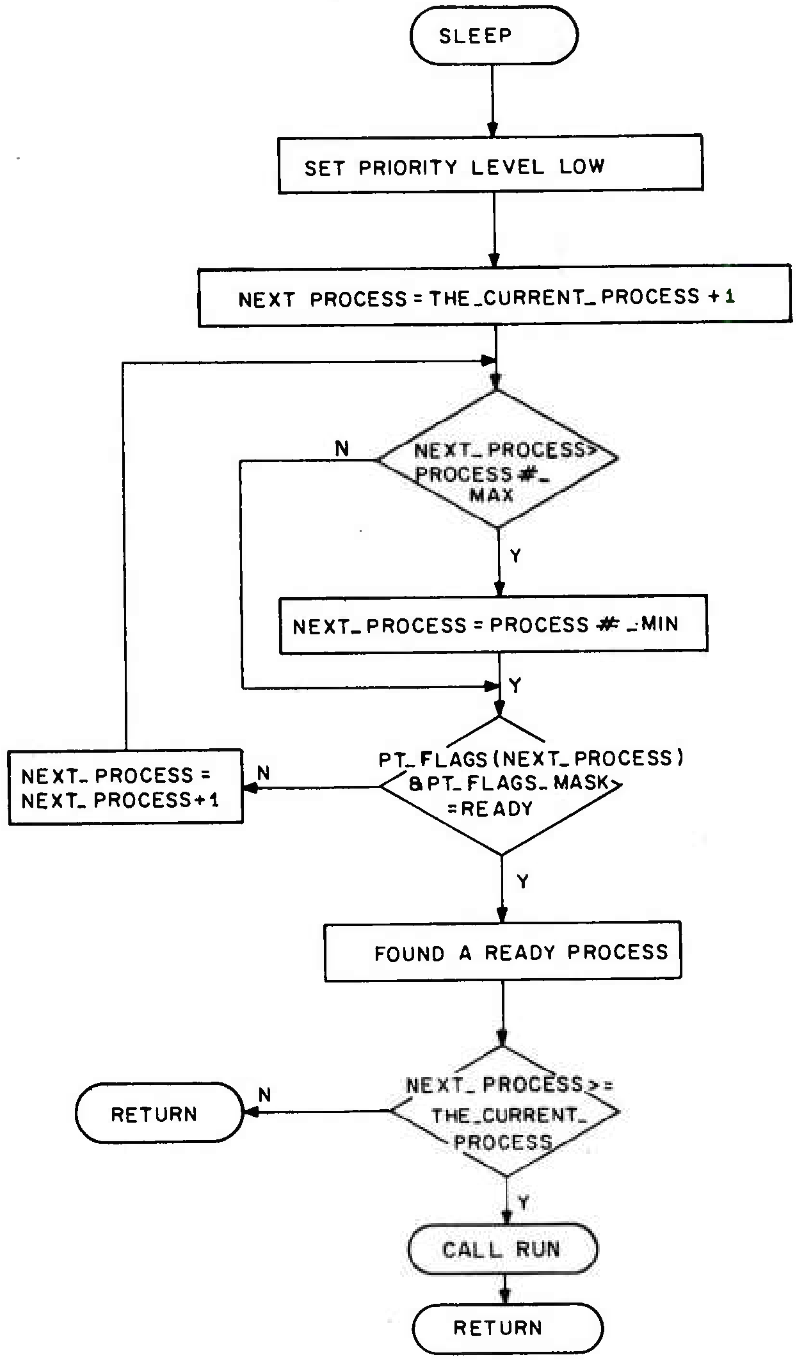
RESCIND
CALL WRITEDIR
DIRECTORY IS ON PROCESSES WAL
N
RETURN ERROR
SEARCH FOR SPECIFIED ACLE
INDEX = DIR_ACL_HEAD (OFFSET)
INDEX=Ø
Y
END OF LIST
RETURN ERROR
N
USER = ACL_USER (INDEX) & ACL_USER_MASK & PROJECT = ACL_PROJECT (INDEX)
N
SAVE LAST = INDEX
INDEX = ACL_CHAIN (INDEX)
Y
SPECIFIED ACLE FOUND
REMOVE FROM CHAIN
INDEX= DIR_ACL_HEAD (OFFSET)
N
REMOVE ELEMENT FROM CHAIN
ACL_CHAIN (SAVE_LAST) =
ACL_CHAIN (INDEX)
Y
REMOVE ELEMENT FROM CHAIN
DIR_ACL_HEAD (OFFSET) =
ACL_CHAIN (INDEX)
PUT ELEMENT AT HEAD OF CHAIN
ACL_CHAIN (INDEX)=ACL_CHAIN (Ø)
ACL_CHAIN (Ø) = INDEX
CALL SOADD
RETURN OK

IB-50,278

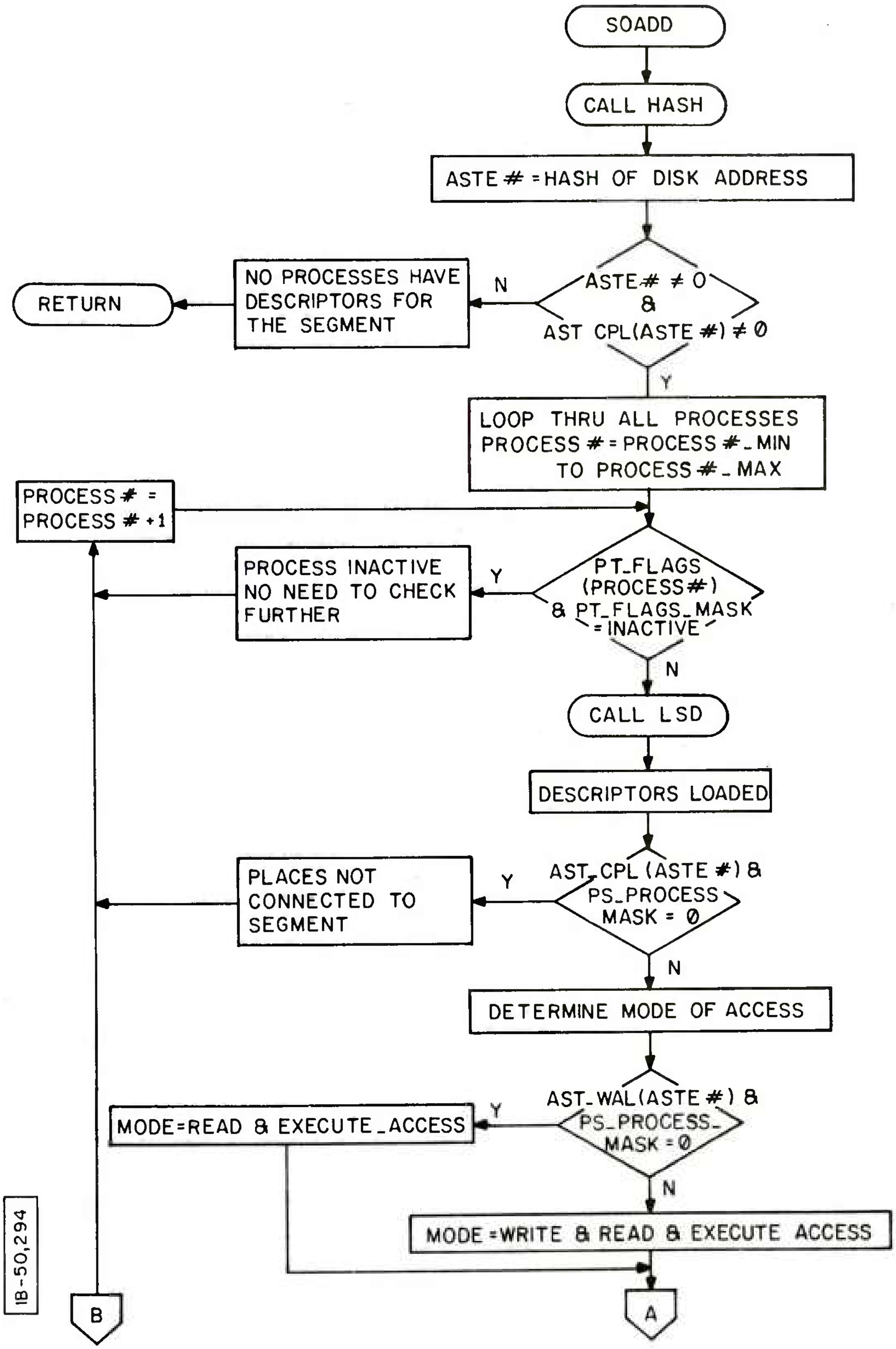

RUN
SET PRIORITY LEVEL HIGH
LOOK FOR CHANGES
REG # = Ø TO REG_MAX
N
X = REG #
REG # >
CROSS REG #
Y
X=REG # + REG_CONSTANT
CHANGE BIT
NOT SET
N
SDR(X) &
SDR_CHANGE_
MASK=
SDR_CHANGED
Y
PERFORM 2 ARITHMETIC SHIFTS RIGHT
VAR=MAIN MEMORY ADDRESS
CHANGE BIT IN
ASTE NOT SET
N
VAR<END
_BLOCK
Y
VAR = MBT_ ASTE_(VAR)
AST_ CHANGE (VAR)= AST_CHANGE (VAR)
I AST_ CHANGED
N
REG # =
REG #_MAX
Y
SAVE KSR1
PT_KSAR1 (THE_CURRENT_PROCESS)=KSAR1
PT_KSDR1 (THE_CURRENT_PROCESS)=KSDR1
CALL LSD
A
IB-50,296

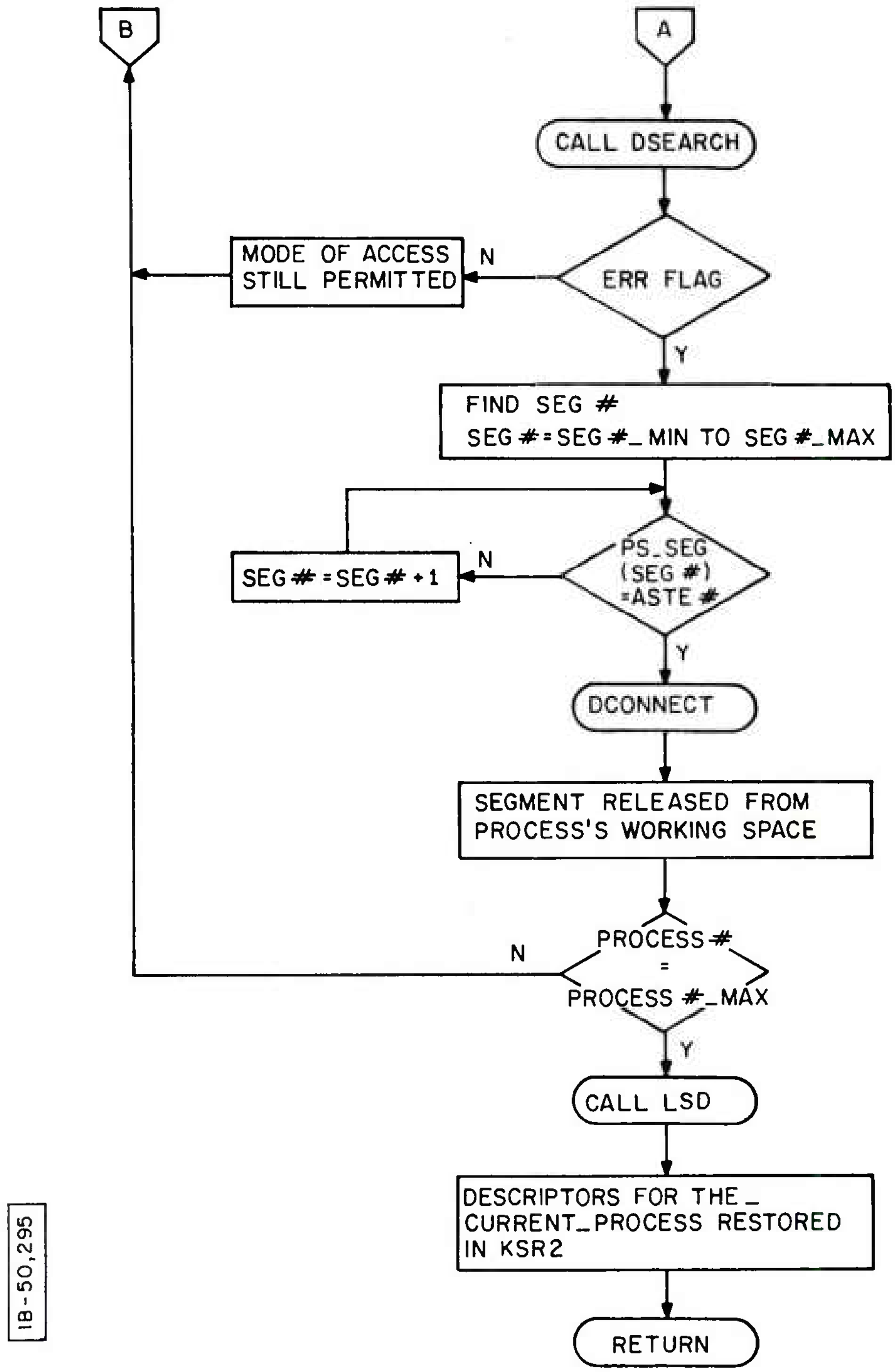

A
DESCRIPTORS FOR NEXT PROCESS LOADED
CALL SWAP
CONTINUE THIS PROCESS
SET PRIORITY LEVEL LOW
RETURN

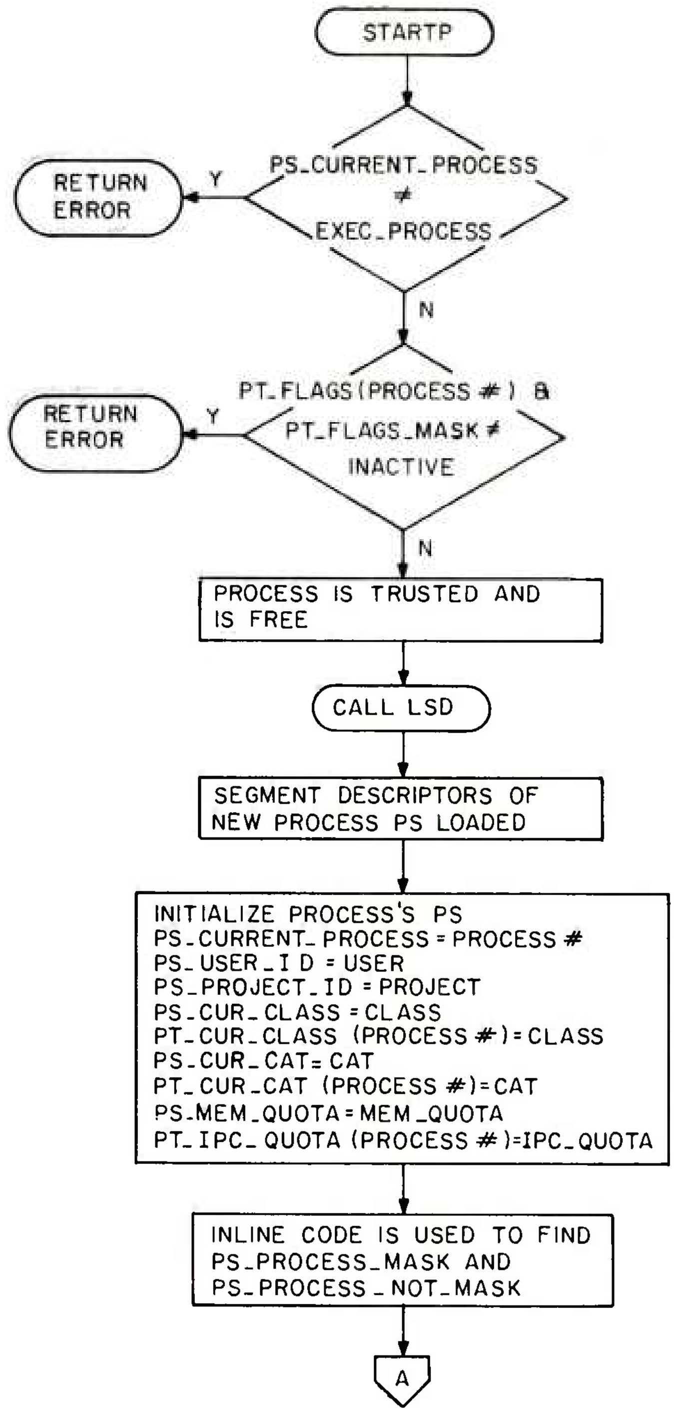
IB-50,297


SLEEP
SET PRIORITY LEVEL LOW
NEXT PROCESS = THE_CURRENT_PROCESS + 1
NEXT_PROCESS> PROCESS #_ MAX
N
Y
NEXT_PROCESS = PROCESS # _MIN
Y
PT_FLAGS (NEXT_PROCESS) & PT_FLAGS_MASK> =READY
N
NEXT_PROCESS = NEXT_PROCESS+1
Y
FOUND A READY PROCESS
NEXT_PROCESS>= THE_CURRENT_PROCESS
N
RETURN
Y
CALL RUN
RETURN

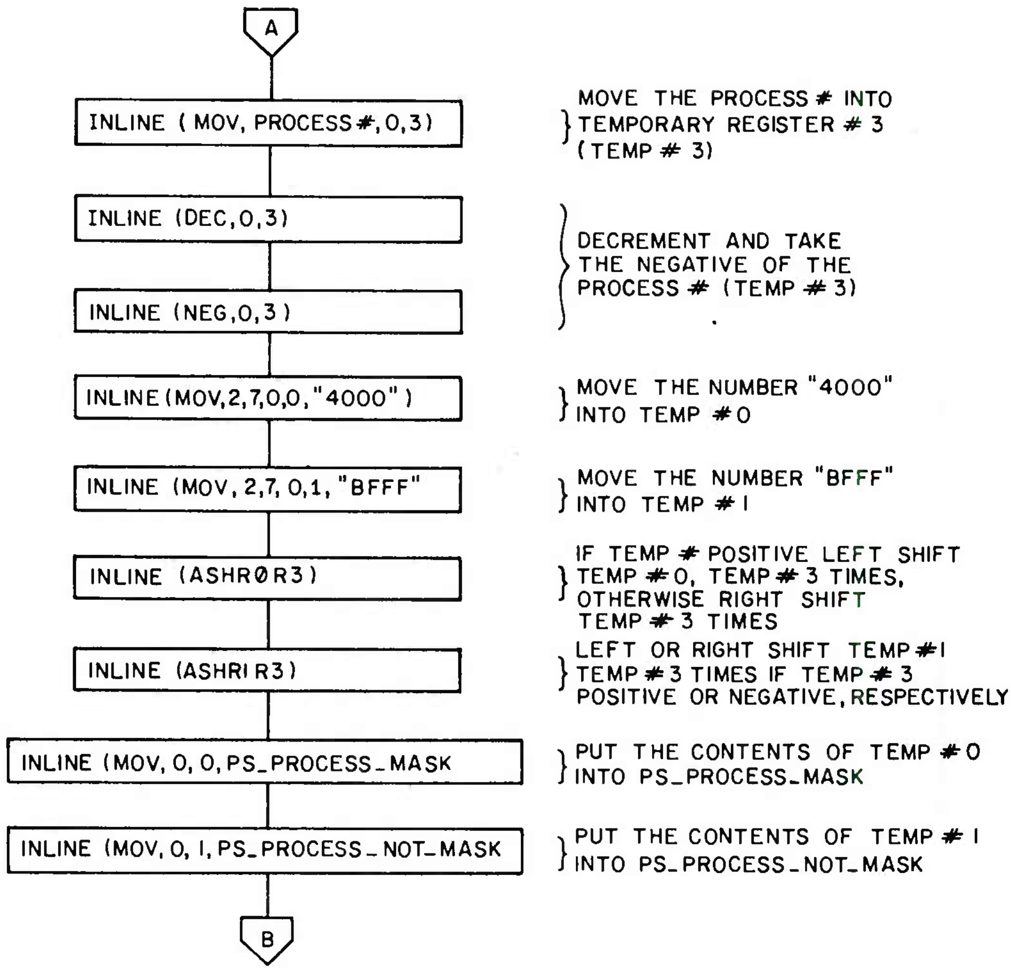
18-50,266


SOADD
CALL HASH
ASTE # =HASH OF DISK ADDRESS
ASTE # ≠ 0 & AST CPL(ASTE #) ≠ Ø
N
NO PROCESSES HAVE DESCRIPTORS FOR THE SEGMENT
RETURN
Y
LOOP THRU ALL PROCESSES PROCESS # = PROCESS #_MIN TO PROCESS #_MAX
PROCESS # = PROCESS # +1
PT_FLAGS (PROCESS #) & PT_FLAGS_MASK = INACTIVE
Y
PROCESS INACTIVE NO NEED TO CHECK FURTHER
N
CALL LSD
DESCRIPTORS LOADED
AST_CPL (ASTE #) & PS_PROCESS MASK = Ø
Y
PLACES NOT CONNECTED TO SEGMENT
N
DETERMINE MODE OF ACCESS
AST_WAL(ASTE #) & PS_PROCESS_ MASK = Ø
Y
MODE=READ & EXECUTE_ACCESS
N
MODE =WRITE & READ & EXECUTE ACCESS
B
A
IB-50,294

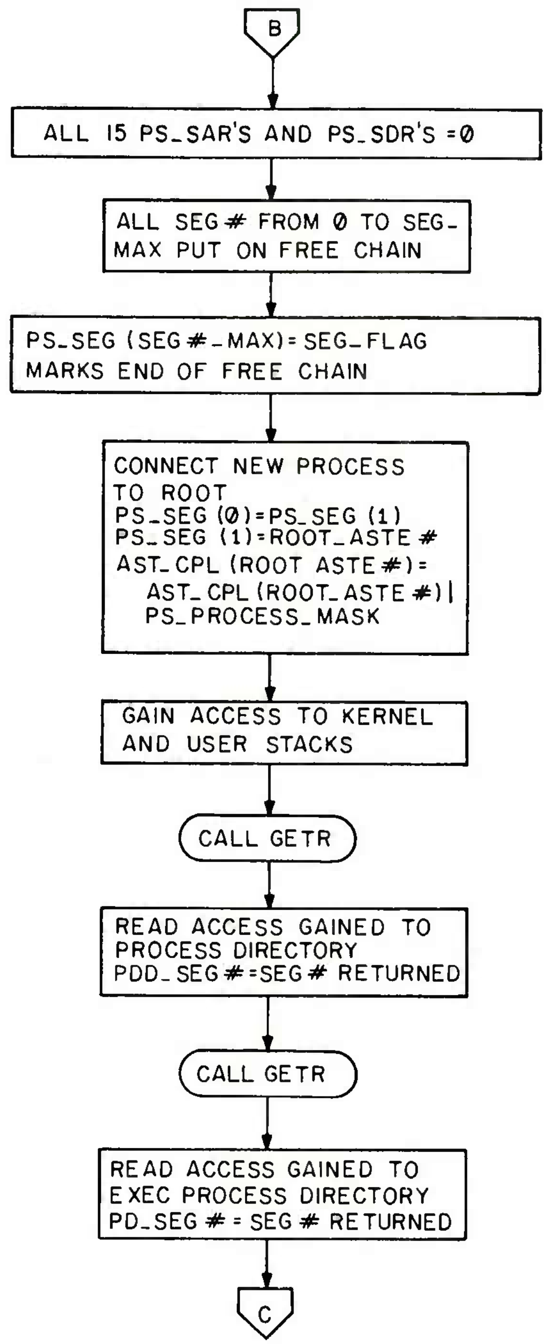

B
A
CALL DSEARCH
ERR FLAG
N
MODE OF ACCESS STILL PERMITTED
Y
FIND SEG #
SEG # = SEG #_MIN TO SEG #_MAX
PS_SEG (SEG #) =ASTE #
N
SEG # = SEG # + 1
Y
DCONNECT
SEGMENT RELEASED FROM PROCESS'S WORKING SPACE
PROCESS # = PROCESS #_MAX
N
Y
CALL LSD
DESCRIPTORS FOR THE_CURRENT_PROCESS RESTORED IN KSR2
RETURN
1B-50,295


STARTP
PS_CURRENT_PROCESS ≠ EXEC_PROCESS
Y
RETURN ERROR
N
PT_FLAGS(PROCESS #) & PT_FLAGS_MASK ≠ INACTIVE
Y
RETURN ERROR
N
PROCESS IS TRUSTED AND IS FREE
CALL LSD
SEGMENT DESCRIPTORS OF NEW PROCESS PS LOADED
INITIALIZE PROCESS'S PS
PS_CURRENT_PROCESS = PROCESS #
PS_USER_ID = USER
PS_PROJECT_ID = PROJECT
PS_CUR_CLASS = CLASS
PT_CUR_CLASS (PROCESS #) = CLASS
PS_CUR_CAT = CAT
PT_CUR_CAT (PROCESS #) = CAT
PS_MEM_QUOTA = MEM_QUOTA
PT_IPC_QUOTA (PROCESS #) = IPC_QUOTA
INLINE CODE IS USED TO FIND PS_PROCESS_MASK AND PS_PROCESS_NOT_MASK
A

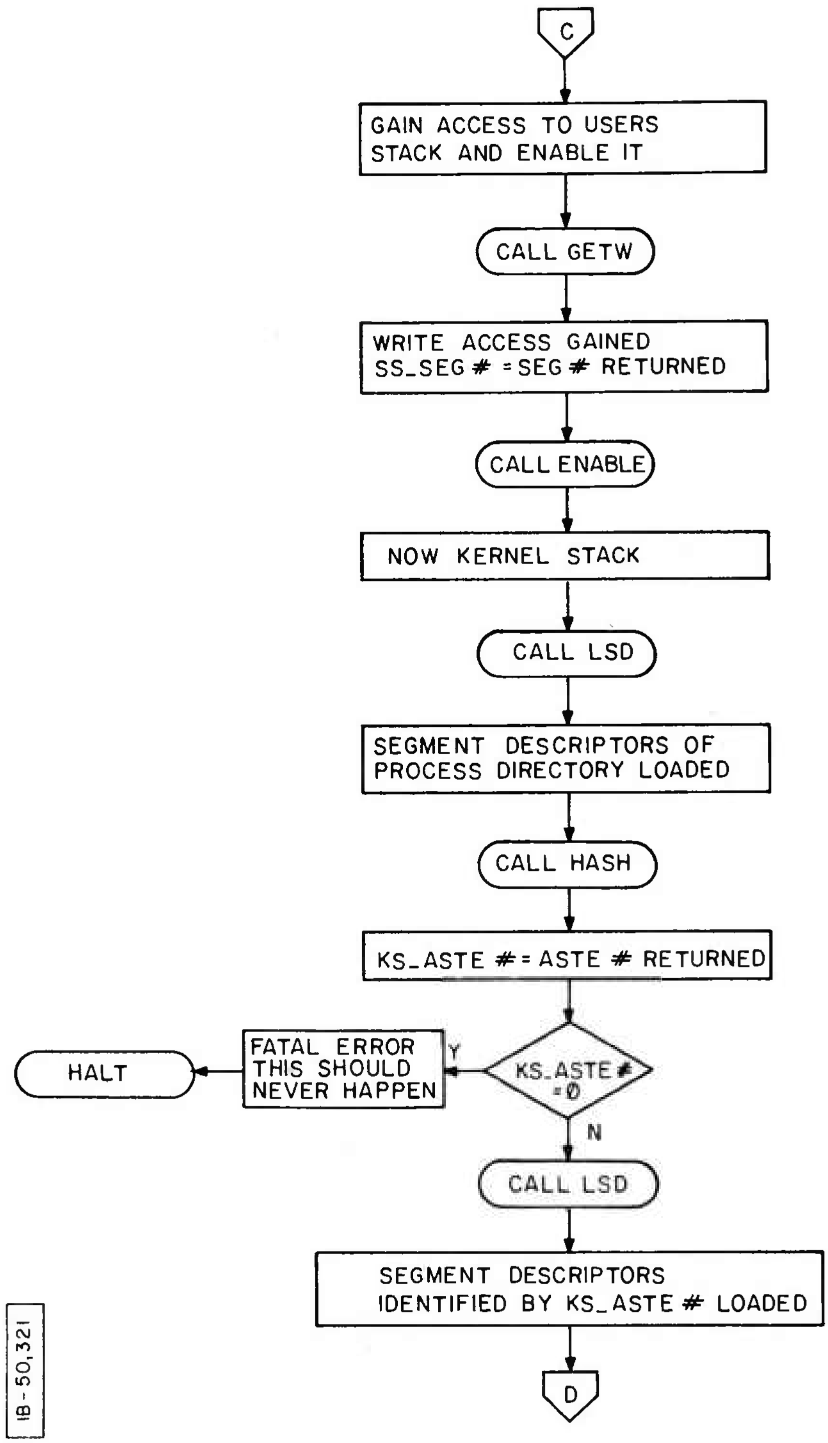
IB-50,318

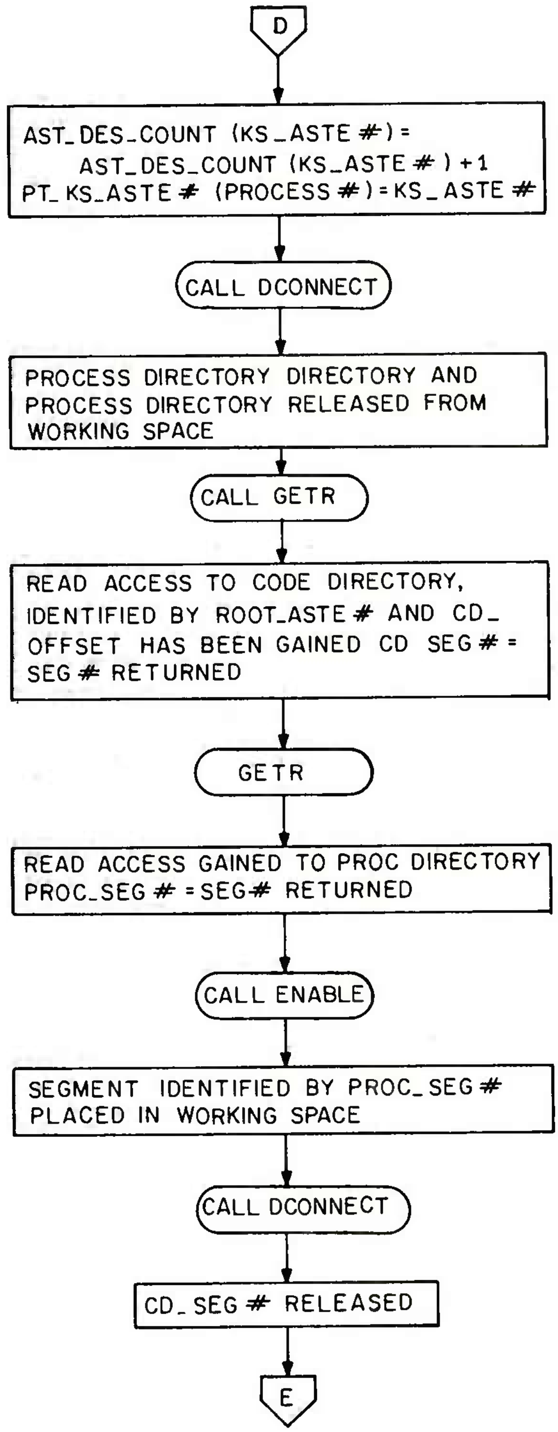

A
INLINE (MOV, PROCESS #, 0,3)
MOVE THE PROCESS # INTO TEMPORARY REGISTER # 3 (TEMP # 3)
INLINE (DEC,0,3)
INLINE (NEG,0,3)
DECREMENT AND TAKE THE NEGATIVE OF THE PROCESS # (TEMP # 3)
INLINE (MOV,2,7,0,0,"4000")
MOVE THE NUMBER "4000" INTO TEMP # 0
INLINE (MOV, 2,7, 0,1, "BFFF"
MOVE THE NUMBER "BFFF" INTO TEMP # 1
INLINE (ASHR0R3)
IF TEMP # POSITIVE LEFT SHIFT TEMP # 0, TEMP # 3 TIMES, OTHERWISE RIGHT SHIFT TEMP # 3 TIMES
INLINE (ASHR1 R3)
LEFT OR RIGHT SHIFT TEMP # 1 TEMP # 3 TIMES IF TEMP # 3 POSITIVE OR NEGATIVE, RESPECTIVELY
INLINE (MOV, 0, 0, PS_PROCESS_MASK
PUT THE CONTENTS OF TEMP # 0 INTO PS_PROCESS_MASK
INLINE (MOV, 0, 1, PS_PROCESS_NOT_MASK
PUT THE CONTENTS OF TEMP # 1 INTO PS_PROCESS_NOT_MASK
B

IB-50,319

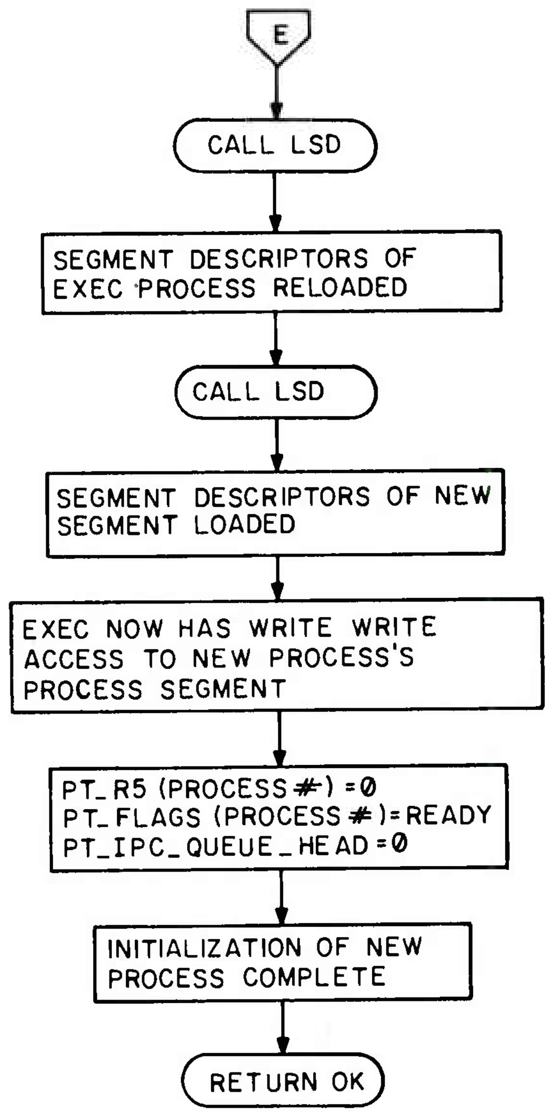

B
ALL 15 PS_SAR'S AND PS_SDR'S = Ø
ALL SEG # FROM Ø TO SEG_MAX PUT ON FREE CHAIN
PS_SEG (SEG #_MAX) = SEG_FLAG MARKS END OF FREE CHAIN
CONNECT NEW PROCESS TO ROOT
PS_SEG (Ø) = PS_SEG (1)
PS_SEG (1) = ROOT_ASTE #
AST_CPL (ROOT ASTE #) = AST_CPL (ROOT_ASTE #) | PS_PROCESS_MASK
GAIN ACCESS TO KERNEL AND USER STACKS
CALL GETR
READ ACCESS GAINED TO PROCESS DIRECTORY
PDD_SEG # = SEG # RETURNED
CALL GETR
READ ACCESS GAINED TO EXEC PROCESS DIRECTORY
PD_SEG # = SEG # RETURNED
C

IB-50,320

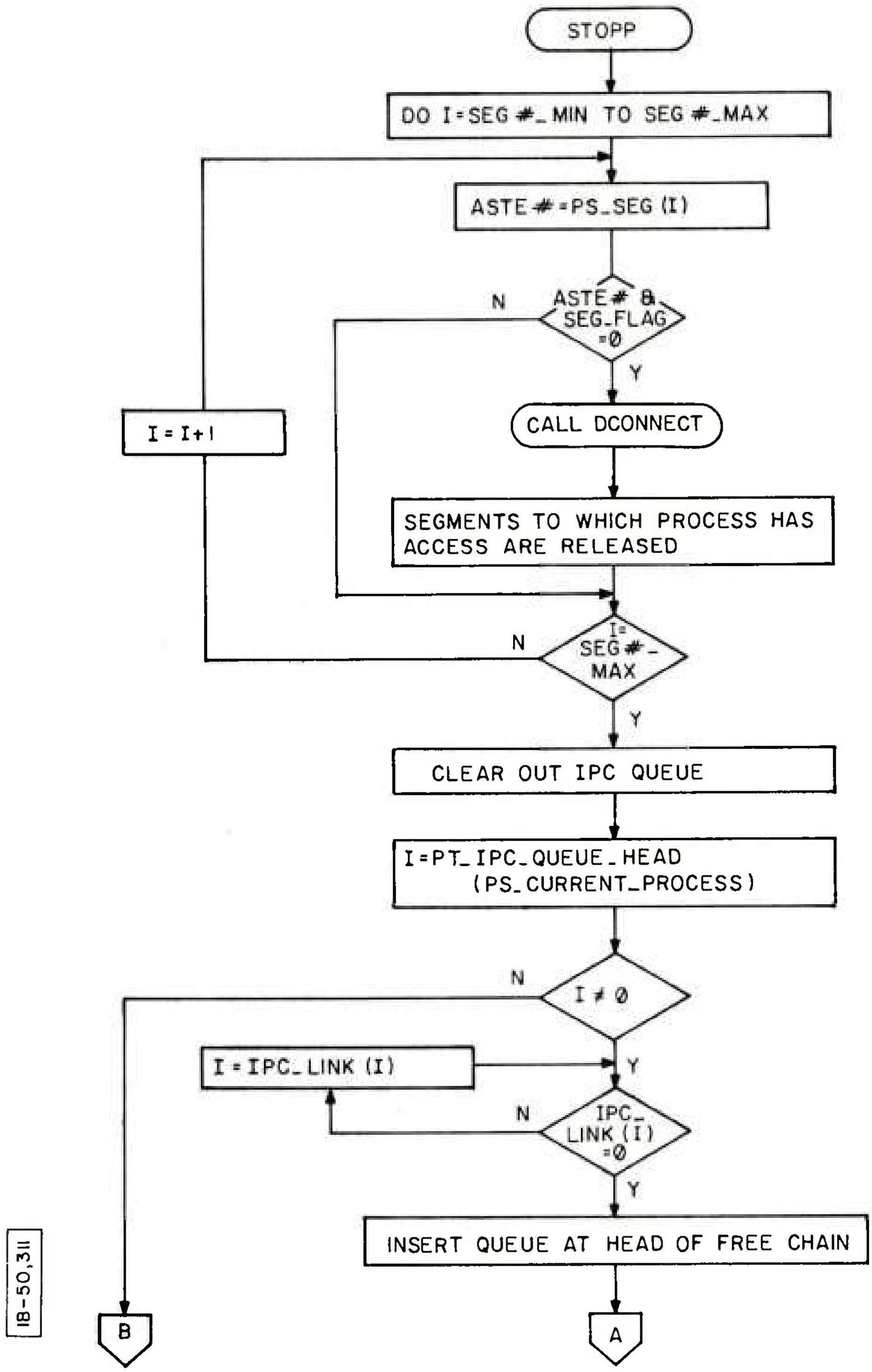

C
GAIN ACCESS TO USERS STACK AND ENABLE IT
CALL GETW
WRITE ACCESS GAINED SS_SEG # = SEG # RETURNED
CALL ENABLE
NOW KERNEL STACK
CALL LSD
SEGMENT DESCRIPTORS OF PROCESS DIRECTORY LOADED
CALL HASH
KS_ASTE # = ASTE # RETURNED
KS_ASTE # = Ø
Y
FATAL ERROR THIS SHOULD NEVER HAPPEN
HALT
N
CALL LSD
SEGMENT DESCRIPTORS IDENTIFIED BY KS_ASTE # LOADED
D
IB-50,321

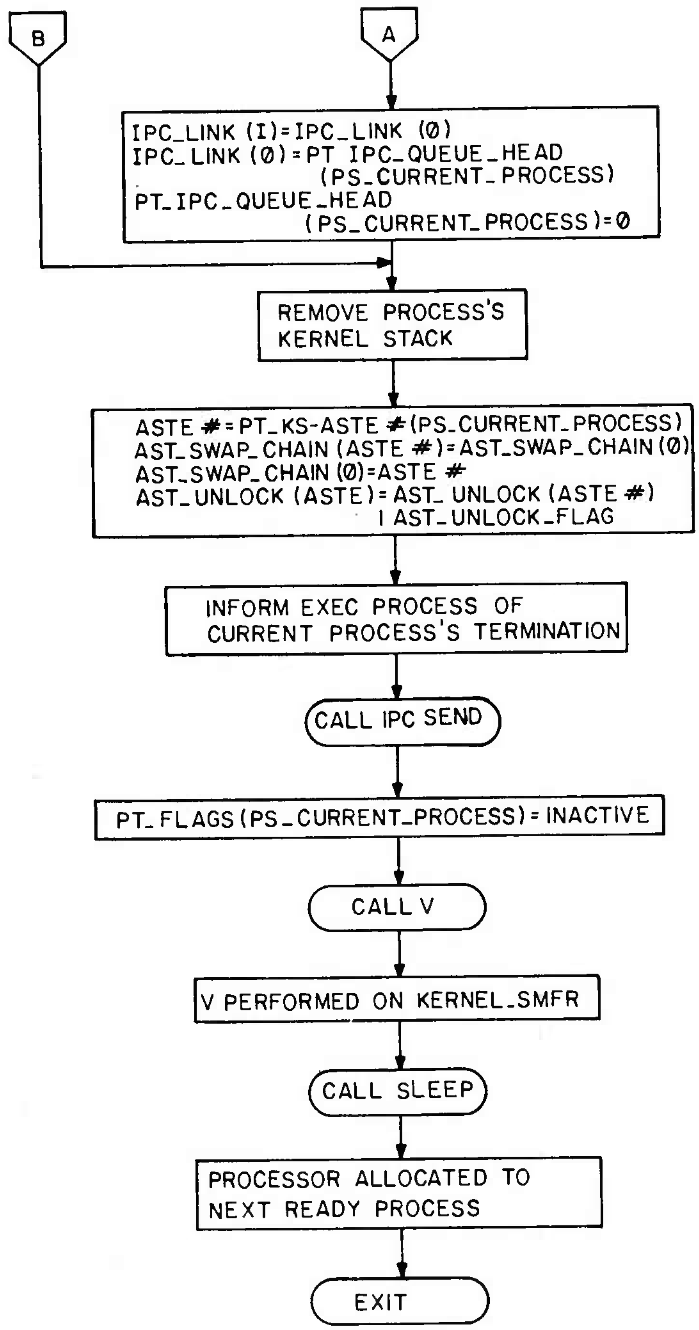

D
AST_DES_COUNT (KS_ASTE #) =
AST_DES_COUNT (KS_ASTE #) + 1
PT_KS_ASTE # (PROCESS #) = KS_ASTE #
CALL DCONNECT
PROCESS DIRECTORY DIRECTORY AND PROCESS DIRECTORY RELEASED FROM WORKING SPACE
CALL GETR
READ ACCESS TO CODE DIRECTORY, IDENTIFIED BY ROOT_ASTE # AND CD_OFFSET HAS BEEN GAINED CD SEG # = SEG # RETURNED
GETR
READ ACCESS GAINED TO PROC DIRECTORY PROC_SEG # = SEG # RETURNED
CALL ENABLE
SEGMENT IDENTIFIED BY PROC_SEG # PLACED IN WORKING SPACE
CALL DCONNECT
CD_SEG # RELEASED
E

IB-50,327

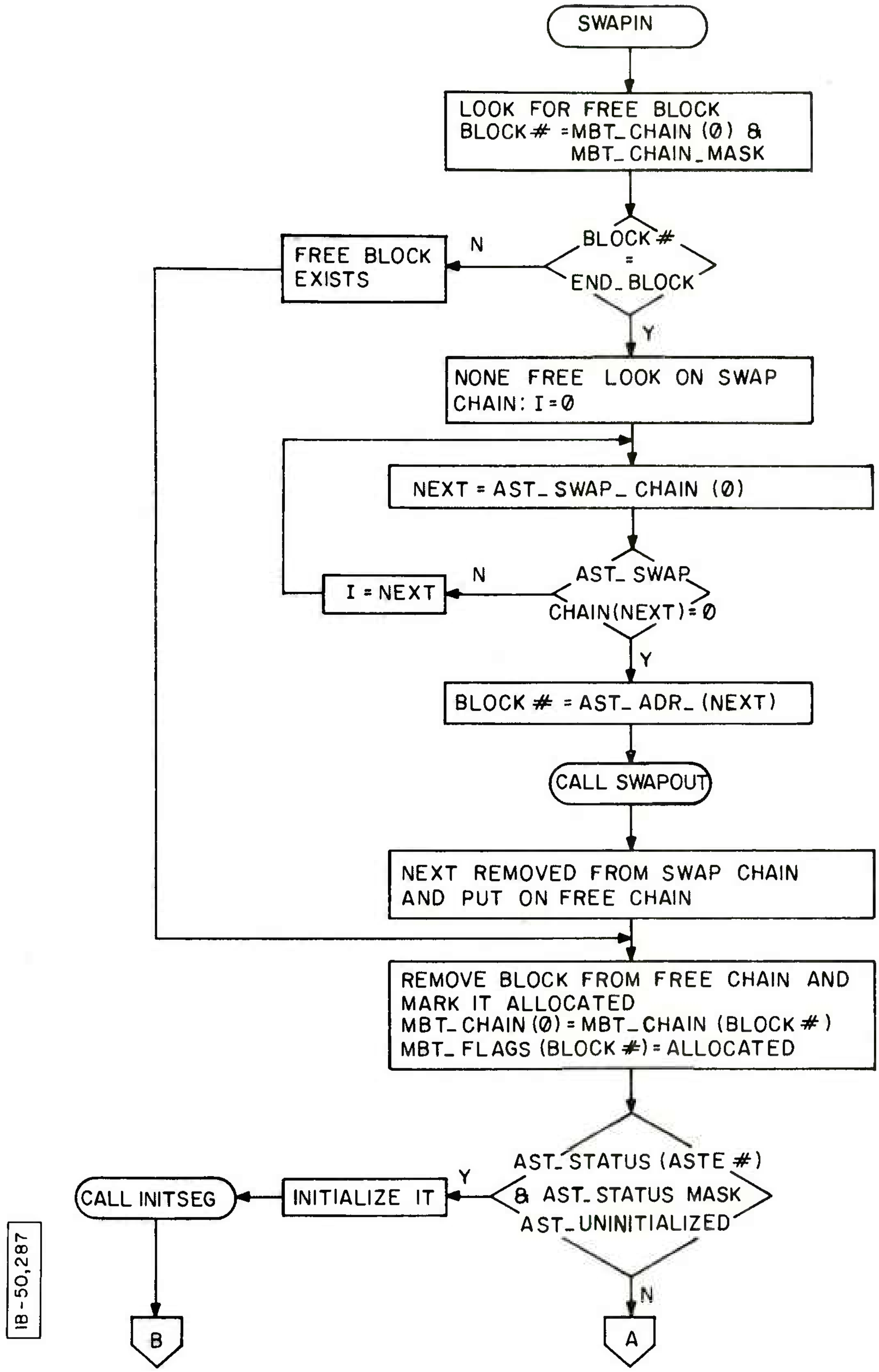

E
CALL LSD
SEGMENT DESCRIPTORS OF EXEC PROCESS RELOADED
CALL LSD
SEGMENT DESCRIPTORS OF NEW SEGMENT LOADED
EXEC NOW HAS WRITE WRITE ACCESS TO NEW PROCESS'S PROCESS SEGMENT
PT_R5 (PROCESS #) = 0
PT_FLAGS (PROCESS #) = READY
PT_IPC_QUEUE_HEAD = 0
INITIALIZATION OF NEW PROCESS COMPLETE
RETURN OK

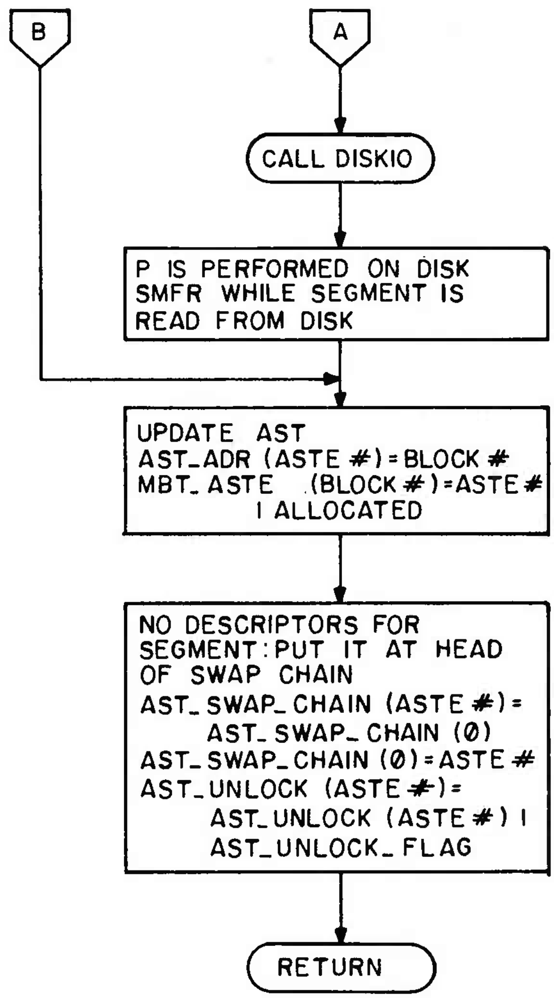
IB-50,322


STOPP
DO I=SEG #_MIN TO SEG #_MAX
ASTE # =PS_SEG (I)
ASTE # & SEG_FLAG =Ø
N
Y
CALL DCONNECT
SEGMENTS TO WHICH PROCESS HAS ACCESS ARE RELEASED
I = I+1
I= SEG #_ MAX
N
Y
CLEAR OUT IPC QUEUE
I=PT_IPC_QUEUE_HEAD (PS_CURRENT_PROCESS)
I ≠ Ø
N
Y
I = IPC_LINK (I)
IPC_ LINK (I) =Ø
N
Y
INSERT QUEUE AT HEAD OF FREE CHAIN
B
A
IB-50,311

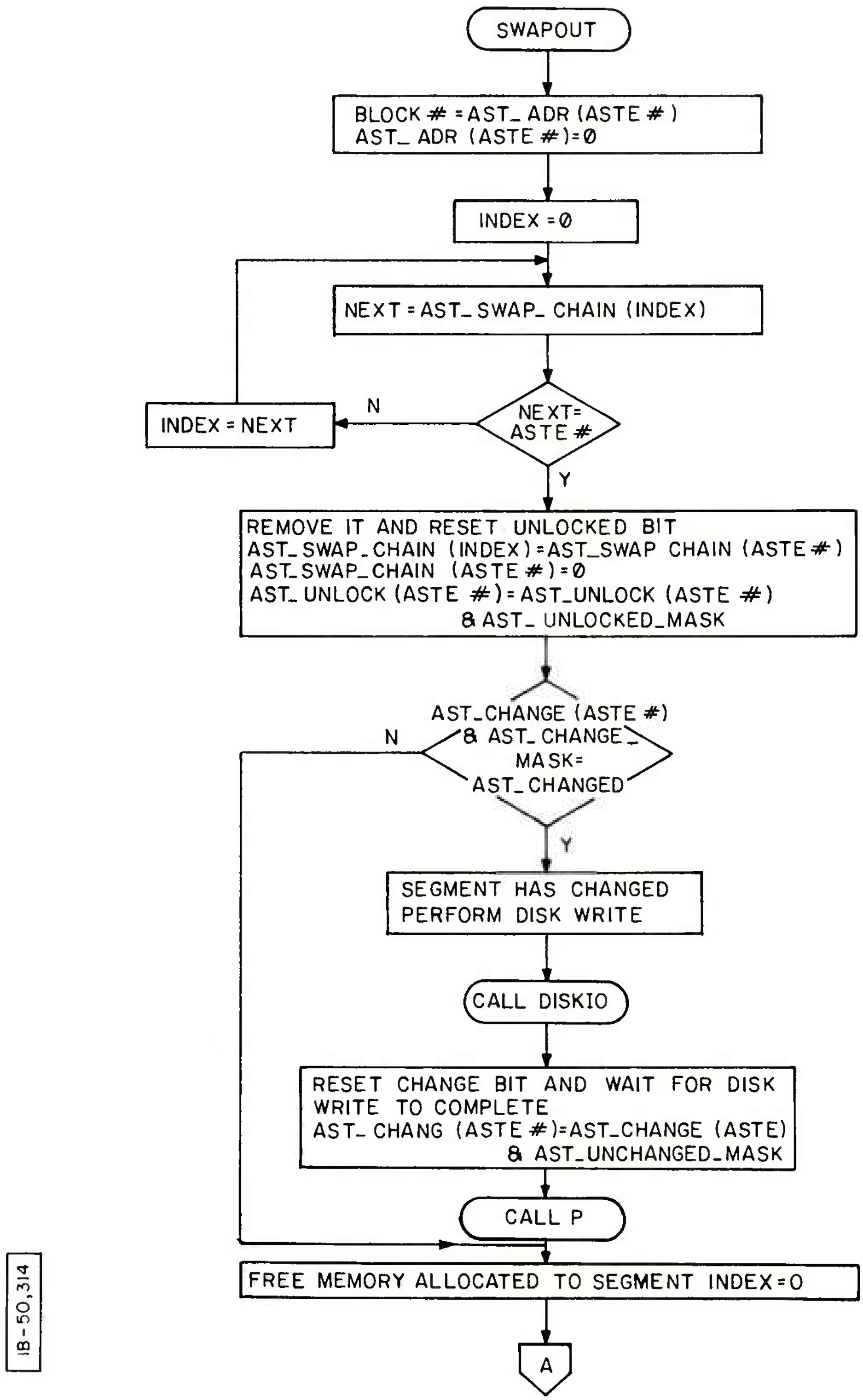

B
A
IPC_LINK (I)=IPC_LINK (Ø)
IPC_LINK (Ø)=PT IPC_QUEUE_HEAD (PS_CURRENT_PROCESS)
PT_IPC_QUEUE_HEAD (PS_CURRENT_PROCESS)=Ø
REMOVE PROCESS'S KERNEL STACK
ASTE #=PT_KS-ASTE #(PS_CURRENT_PROCESS)
AST_SWAP_CHAIN (ASTE #)=AST_SWAP_CHAIN (Ø)
AST_SWAP_CHAIN (Ø)=ASTE #
AST_UNLOCK (ASTE)=AST_UNLOCK (ASTE #) | AST_UNLOCK_FLAG
INFORM EXEC PROCESS OF CURRENT PROCESS'S TERMINATION
CALL IPC SEND
PT_FLAGS (PS_CURRENT_PROCESS)=INACTIVE
CALL V
V PERFORMED ON KERNEL_SMFR
CALL SLEEP
PROCESSOR ALLOCATED TO NEXT READY PROCESS
EXIT

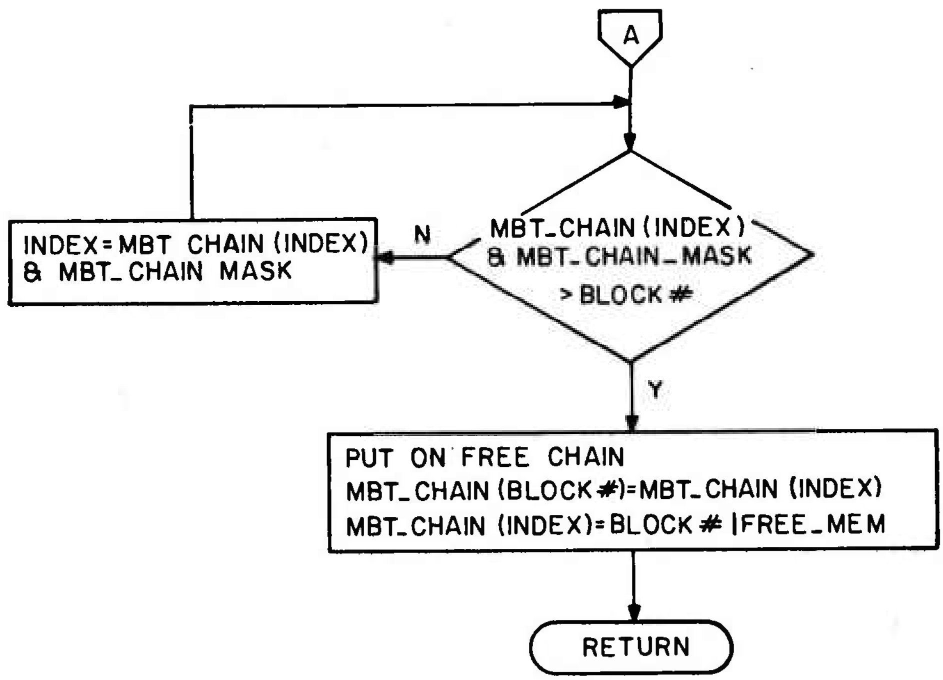
IB-50,312


SWAPIN
LOOK FOR FREE BLOCK
BLOCK # =MBT_CHAIN (Ø) & MBT_CHAIN_MASK
BLOCK # = END_BLOCK
N
FREE BLOCK EXISTS
Y
NONE FREE LOOK ON SWAP CHAIN: I=Ø
NEXT = AST_SWAP_CHAIN (Ø)
AST_SWAP CHAIN(NEXT)=Ø
N
I = NEXT
Y
BLOCK # = AST_ADR_(NEXT)
CALL SWAPOUT
NEXT REMOVED FROM SWAP CHAIN AND PUT ON FREE CHAIN
REMOVE BLOCK FROM FREE CHAIN AND MARK IT ALLOCATED
MBT_CHAIN (Ø) = MBT_CHAIN (BLOCK #)
MBT_FLAGS (BLOCK #) = ALLOCATED
AST_STATUS (ASTE #) & AST_STATUS MASK AST_UNINITIALIZED
Y
INITIALIZE IT
CALL INITSEG
B
N
A
IB-50,287

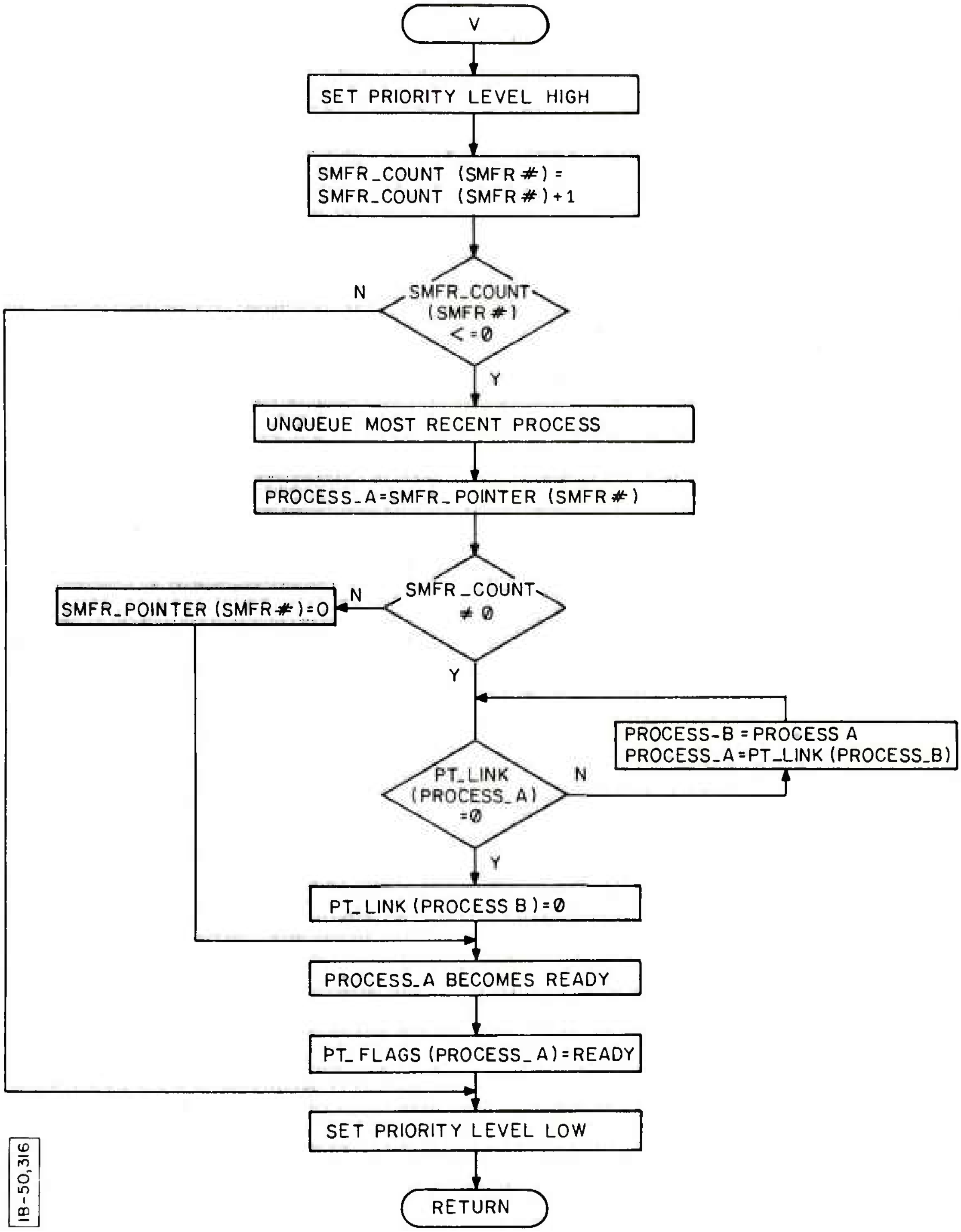

B
A
CALL DISKIO
P IS PERFORMED ON DISK SMFR WHILE SEGMENT IS READ FROM DISK
UPDATE AST
AST_ADR (ASTE #) = BLOCK #
MBT_ASTE (BLOCK #) = ASTE # | ALLOCATED
NO DESCRIPTORS FOR SEGMENT: PUT IT AT HEAD OF SWAP CHAIN
AST_SWAP_CHAIN (ASTE #) = AST_SWAP_CHAIN (Ø)
AST_SWAP_CHAIN (Ø) = ASTE #
AST_UNLOCK (ASTE #) = AST_UNLOCK (ASTE #) | AST_UNLOCK_FLAG
RETURN

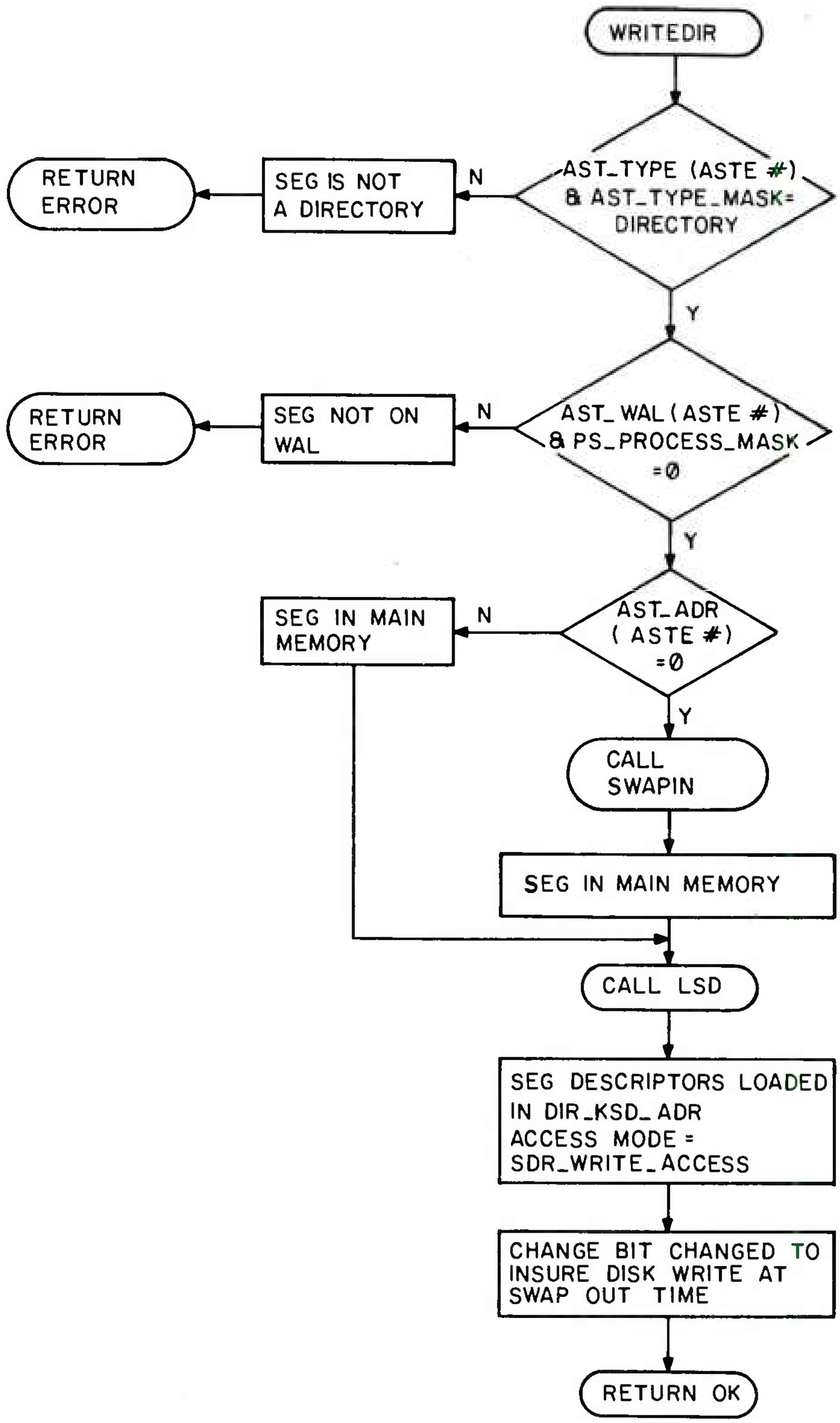
IB-50,288


SWAPOUT
BLOCK # = AST_ADR (ASTE #)
AST_ADR (ASTE #) = 0
INDEX = 0
NEXT = AST_SWAP_CHAIN (INDEX)
NEXT = ASTE #
N
Y
INDEX = NEXT
REMOVE IT AND RESET UNLOCKED BIT
AST_SWAP_CHAIN (INDEX) = AST_SWAP CHAIN (ASTE #)
AST_SWAP_CHAIN (ASTE #) = 0
AST_UNLOCK (ASTE #) = AST_UNLOCK (ASTE #)
& AST_UNLOCKED_MASK
AST_CHANGE (ASTE #)
& AST_CHANGE_
MASK =
AST_CHANGED
N
Y
SEGMENT HAS CHANGED
PERFORM DISK WRITE
CALL DISKIO
RESET CHANGE BIT AND WAIT FOR DISK
WRITE TO COMPLETE
AST_CHANG (ASTE #) = AST_CHANGE (ASTE)
& AST_UNCHANGED_MASK
CALL P
FREE MEMORY ALLOCATED TO SEGMENT INDEX = 0
A
IB-50,314


A
INDEX=MBT CHAIN (INDEX)
& MBT_CHAIN MASK
N
MBT_CHAIN (INDEX)
& MBT_CHAIN_MASK
>BLOCK #
Y
PUT ON FREE CHAIN
MBT_CHAIN (BLOCK #)=MBT_CHAIN (INDEX)
MBT_CHAIN (INDEX)=BLOCK # |FREE_MEM
RETURN

IB-50,315


V
SET PRIORITY LEVEL HIGH
SMFR_COUNT (SMFR #) =
SMFR_COUNT (SMFR #) + 1
SMFR_COUNT (SMFR #) < = Ø
N
Y
UNQUEUE MOST RECENT PROCESS
PROCESS_A = SMFR_POINTER (SMFR #)
SMFR_COUNT ≠ Ø
N
SMFR_POINTER (SMFR #) = 0
Y
PROCESS-B = PROCESS A
PROCESS_A = PT_LINK (PROCESS_B)
PT_LINK (PROCESS_A) = Ø
N
Y
PT_LINK (PROCESS B) = Ø
PROCESS_A BECOMES READY
PT_FLAGS (PROCESS_A) = READY
SET PRIORITY LEVEL LOW
RETURN
IB-50,316


WRITEDIR
AST_TYPE (ASTE #) & AST_TYPE_MASK= DIRECTORY
N
SEG IS NOT A DIRECTORY
RETURN ERROR
Y
AST_WAL (ASTE #) & PS_PROCESS_MASK =Ø
N
SEG NOT ON WAL
RETURN ERROR
Y
AST_ADR (ASTE #) =Ø
N
SEG IN MAIN MEMORY
Y
CALL SWAPIN
SEG IN MAIN MEMORY
CALL LSD
SEG DESCRIPTORS LOADED IN DIR_KSD_ADR ACCESS MODE = SDR_WRITE_ACCESS
CHANGE BIT CHANGED TO INSURE DISK WRITE AT SWAP OUT TIME
RETURN OK

IB-50,276